Theories for Session-based Governance for Large-scale Distributed Systems

Tzu-chun Chen

Department of Computer Science

Queen Mary, University of London

A thesis submitted for the degree of

PhilosophiæDoctor (PhD)

2013, March

Supervised by: Dr Kohei Honda

Abstract

Large-scale distributed systems and distributed computing are the pillars of IT infrastructure and society nowadays. Robust theoretical principles for designing, building, managing and understanding the interactive behaviours of such systems need to be explored. A promising approach for establishing such principles is to view the session as the key unit for design, execution and verification.

Governance is a general term for verifying whether activities meet the specified requirements and for enforcing safe behaviours among processes. This thesis, based on the asynchronous π-calculus and the theory of session types, provides a monitoring framework and a theory for validating specifications, verifying mutual behaviours during runtime, and taking actions when non-compliant behaviours are detected. We explore properties and principles for governing large-scale distributed systems, in which autonomous and heterogeneous system components interact with each other in the network to accomplish application goals.

This thesis, incorporating lessons from my participation in a substantial practical project, the Ocean Observatories Initiative (OOI), proposes an asynchronous monitoring framework and the process calculus for dynamically governing the asynchronous interactions among distributed multiple applications. We prove that this monitoring model guarantees the satisfaction of global assertions, and state and prove theorems of local and global safety, transparency, and session fidelity. We also study and introduce the semantic mechanisms for runtime session-based governance and the principles of validation of *stateful* specifications through capturing the runtime asynchronous interactions.

Acknowledgements

My greatest thanks go to my supervisor Prof. Kohei Honda, who has been without doubt the person with the greatest influence in my life. He constantly encouraged and supported me not only for my doctoral research but also for my living in London. He advised me almost every day, tirelessly, dedicating more time and effort than anyone could ask from a supervisor. Because of his great supports, I had an invaluable PhD life. He was always cheerful, approachable, supportive, and put my doctorate studies as his first priority. For me, he is the model of a great computer scientist. He has endless loves and enthusiasm for computer science. He is so dedicated to it and so willing to communicate with people and to listen to different opinions. Kohei, thank you so much. Thank you for your cares, supports, and advices on both professional and personal issues. There are no words to express the full measure of my gratitude. How lucky I am to know you, to be your PhD student, and to share all the happy moments with you in the past few years.

Also, my warmest thanks go to my second supervisor, Prof. Nobuko Yoshida (Imperial College London). I learned from Nobuko both as a student and a co-author. She has a mind of steal and, at the same time, has the softest and warmest heart. After Kohei's suddent passed away, she supported me and worked extremely hard to help me to complete my thesis even though herself is in a great sadness. Thank you Nobuko. Thank you so much for your valuable advices, efforts, and cares. Kohei and you both give me great impacts on my research life.

My thanks also go to Prof. Edmund Robinson (Queen Mary, University of London), who gave me the precious help to proof read the thesis, especially when he, as the head of the department, was extremely busy. Thank you

Edmund. Thank you for your patience and encouragements to help me go through the difficult period.

I would like to express my gratefulness that I have worked with, and learned from, Dr. Romain Demangeon (Queen Mary, University of London), Dr. Luca Fossati (Queen Mary, University of London), Dr. Laura Bocchi (University of Leicester), Dr. Pierre-Malo Deniélou (Royal Holloway, University of London), and Dr. Raymond Hu (Imperial College London). Thank you so much for your helps and guidances. Especially, thank you Romain, Laura and Luca. Your advices and comments make this thesis completed.

Last but not least, my special thanks go to my families, my beloved parents, my dearest brother Chun-ting and cousin Yuju, and my friends Dr. Muyan Wang, Dr. Tzu-ching Horng, Pelin Yilmaz, Yvonne Chi, Dr. Kurosh Meshkat, Dr. Yuk-Chien Liu, Faris Abou-Saleh, and all of my colleagues. Thank you for accompanying me to overcome the most difficult time after the sudden death of Kohei, a great sadness for all people who know him. Before this, I did not know how important and how precious my families, my loved ones, and my friends are. You made me understand how lucky I am. Thank you all. You are all my angels.

Most important, thanks to the funding body of EPSRC grants EP/G015481/1 and Queen Mary, University of London. Without your support, I cannot even have the chance to study abroad to pursue a doctoral degree and to know those people who are so important and so valuable for me.

I dedicate this work to my parents. I believe there are no parents like them devoting their loves to their children so much. Without their loves and unconditional supports, I cannot finish this thesis, and I cannot have the chance to continue my research works.

I also dedicate it to my supervisor Prof. Kohei Honda, who died of a stroke on Thuesday 4 December 2012. I still have not taken it in, and I think I will never do. Kohei, you had promised me that I will have a great adventure of computer science. And I promise you that I will continue it.

Contents

List of Figures

1

Introduction

1.1 Motivations and Objectives

Motivations Governance is the collection of activities of governing. The purpose of governance for a large system is to ensure that independent system components (e.g. system agents, normal users, endpoint devices, etc.) can securely achieve their respective goals and, at the same time, the overall performance of the system is maintained and consistent. The related topics for achieving this purpose include access control mechanisms (e.g. access control matrix (105), DAC, MAC (53), and RBAC (62, 141), etc.), information flow control (28, 29, 65, 126, 140, 162), and policy languages for describing laws and regulations or specifying contracts and commitments (17, 54, 144).

Over the last two decades, we have witnessed computing systems evolve from isolated machines to network environments, from the centralised to the distributed; such evolution gives systems the benefits of performing tasks through densely coordinated actions among multiple heterogeneous components with distributed resources (19, 74, 112, 127). Communication in distributed components is becoming crucial for building large-scale applications, aided by a rapidly developing infrastructural support for portable distributed devices through technologies such as cloud computing, messaging and distributed stores. While distribution leads to such virtues as scalability, sharing and resilience (42, 125), specifying and ensuring the diverse correctness properties that individual applications demand poses new technical challenges.

1. INTRODUCTION

In order to understand and formalise interactive behaviours among communicating processes, robust theoretical foundations for such systems are needed. Without capturing the interactive behaviours, we can not judge the safety and reliability of target applications, and hence can not even set the baseline that our governance systems must meet. The traditional framework of sequential computing, such as objects and functions, does not enable us to effectively apprehend the behaviours and properties of distributed computing as a whole, since the central mode of computation in this new context is *communication* rather than, for example, assignment and function application.

In the concrete engineering setting of a large-scale distributed system, for example, the large IT infrastructure for ocean sciences (129), which is our leading example, applications are predominantly built as *asynchronous* interactions among heterogeneous distributed components. To cater for scalability and flexibility (23), it must be possible for each component to be implemented in a different programming language, and likewise for some of its components to be contributed by a third party, and these may be buggy or untrusted; moreover, it is also possible that the properties of services provided by contributors are dynamically discovered and changed. With the issues mentioned above and the problems addressed in (152, 153, 154), foreign components may not have been statically checked and runtime behavioural checks must be enforced.

In reality, most modern large-scale systems are distributed, but their governance relies on ad-hoc monitoring, complex safety enforcement mechanisms, centralised protocols, and expensive testing. And they still fail. Therefore, there is both a practical and theoretical need for the dynamic adjustment of a global specification in order to protect endpoints from malicious attacks and, at the same time, prevent them from producing malicious messages that pollute the network. Session-based governance, in which session types direct the computing, promises to offer a completely distributed, asynchronous governance framework for real-life systems, from fine-grained capability-passing π-calculus, to the semantics of stateful specifications.

Objectives This thesis aims to identify principles to harness the complexity of asynchronous interactions among distributed heterogeneous components through capturing processes' runtime behaviours. With this purpose in mind, we propose the formalism of a *session-based dynamic-asynchronous monitoring* framework:

1. The framework consists of *external* observers or monitors. The prototype of a monitor adopted in this thesis is similar to the one defined in (11, 103, 104). A monitor acts as an endpoint guard, rather than a central mediator.

2. The observers or monitors in the framework *asynchronously* and *distributedly* verify interactions among endpoints, which may come from heterogeneous domains.

3. The monitoring and verification mechanisms of this framework ensure, during runtime, the safe interactions of endpoints in a session through session-type-based local specifications, which are obtained by projection from global protocol specifications.

This framework is general enough to represent most large-scale distributed systems. With this framework, we introduce and prove *governance theories* for safety, transparency, session-fidelity, and the validation of stateful specifications.

One critical issue is how to deal with *asynchrony*. Two difficulties arise when this framework is applied. The first comes from adopting external monitors, which is common in practice, because inserting system internal monitors at every endpoint is expensive and they might be polluted by malicious local processes. This setting however results in *asynchronous interactions* between endpoints and their corresponding monitors since one monitor may economically guard more than one endpoint process. To deal with asynchrony, the mechanism of *permutation* is proposed and formalised.

The second difficulty comes from the desire to specify the endpoint state in specifications. A monitor, for guarding interactions, relies on a specification as a judgement base when it evaluates whether local process behaviour fulfill the intended requirements through observing the asynchronous messages passing from one process to another. In large-scale distributed systems, the use of *state* is omnipresent in specifications to capture real-life scenarios, where the (expected) states of participants in applications, such as the credit of a client for on-line shopping or the booking number for a transaction, play a critical role in specifications. A specification language for dynamic observations becomes non-trivial when *states* and asynchrony are both counted. An issue in the *semantics* of a specification language arises. The issue is that a monitor may be in asynchronous communication with the endpoint it monitors, and some interactions may update the endpoint's state. For example, assume a local specification specifies that a ticket allocation server, with a state called `counter`, should assign a fresh ticket whose

value is equal to the value of counter for a client's request and the value of counter is increased by one after every assignment. Thus, the server is required to assign ticket numbers as a sequence 1,2,3,4,5,.... Intuitively, the local specification of the server can assert that the output ticket value, say x, should be equal to the current value of state counter i.e. $x = \mathsf{counter}$, and can stipulate the update of state as $\mathsf{counter} := \mathsf{counter} + 1$ once an output takes place. We call this kind of specification a *stateful* specification. It seems natural that a monitor can use such *stateful* specification to judge if the server manages state counter according to the specification. However, when the interaction between a monitor and its corresponding local process is asynchronous, the monitor may see ticket assignments out of order and hence some *stateful* specifications, particularly the one in the above example, may lead a monitor to a wrong judgement in determining whether a local process is well-behaved or not. This issue poses a fundamental challenge to the endeavour to provide a consistent specification-verification framework.

In summary, this thesis presents the following contributions for governance of large-scale distributed systems:

1. We provide dynamic monitoring of runtime endpoint behaviour and prove that it guarantees the satisfaction of global specifications.

2. We state and prove the following properties:

   - *Dynamic global safety assurance* ensuring endpoint communication conformance (communication safety in (86)) and stating that a fully monitored network behaves well with respect to the given global specification.
   - *Dynamic global transparency* (43, 143) ensuring that if every endpoint program conforms to its local specification, the presence of monitors do not change the global interactions.
   - *Session fidelity* (86) ensuring that interactions in a session always follow stipulated global specifications step by step.

3. We introduce the semantics of specifications for dynamic adjustment and prove the properties of satisfaction and validation of *stateful* specifications for a formalism.

Ongoing and future work addresses the development of policy languages and the properties for commitments.

1.2 Contributions

This dissertation focuses on the theories of dynamic distributed monitoring and specifying stateful specifications. It covers a wide range of specifications and semantics, while presenting a complementary approach for runtime verification. The main contributions of this thesis are divided into three parts:

Capability-passing π-calculus. This part is covered in Chapter 5, which contributes:

1. The introduction of a multi-party session-based π-calculus with new capability passing primitives for fine-grained control of endpoint behaviour through *session creation*.
2. The presentation of an exact semantic account of how a session can be initiated linearly with our calculus.
3. The introduction of a framework with an accurate view of the reality of distributed systems (15) through formalising session-roles, name-passing, and asynchronous messaging.
4. The presentation of an abstraction of asynchronous operational semantics for both endpoint processes (locally) and the network (globally).

Dynamic distributed monitoring. This part is covered in Chapters 6 and 7, which contribute:

1. A model of dynamic *asynchronous* monitoring for distributed systems featuring the following elements:
   - The endpoints which are interacting with one another may come from heterogeneous domains, each of which may have different data structure, interface, typing compiler, or authentication policies. The endpoint code contributed by a heterogeneous domain may be written in a different language or possibly malicious.
   - The external monitors, each of which may guard more than one endpoint, protect endpoints from malicious attacks in the network and, at the same time, prevent endpoints from polluting the network. This practical setting results in asynchronous interactions between endpoints and their corresponding monitors.

- The global specifications (22) enable concise application-level multi-party protocols, and local monitoring guarantees overall network conformance to stipulated global protocols at runtime.

2. The required permutation mechanism for external monitors. Due to asynchrony, the order of messages is not preserved during runtime; the permutation mechanism is therefore needed for monitors to capture the real-world asynchronous messagings.
3. The formalisation and establishment of properties of the monitored network, including:
   - *Local and global safety.* Local safety (Theorem 7.4.3) states that a monitored process always behaves well with respect to the specification. Global safety (Theorem 7.4.16) states that a fully monitored network behaves well with respect to the given global specifications.
   - *Local and global transparency* (43, 143). Local transparency (Theorem 7.4.5) states that a monitored process behaves as an unmonitored process when the latter is well-behaved (e.g., it has been statically checked). Global transparency (Theorem 7.4.17) states that a monitored network and an unmonitored network have equivalent behaviour when the latter is well-behaved with respect to the same collection of local specifications, which are projected from well-formed global specifications.
   - *Session fidelity* (86) (Theorem 7.4.28) states that, if all session message exchanges in a monitored/unmonitored network behave well with respect to the specifications (as communications dynamically unfold), then this network exactly follows the original global specifications.
   - *Conformance with respect to the global observables* (Proposition 7.5.7) states that, whenever the global observable behaviours are coherent, as every possible endpoint action takes place, the resulting global observable behaviours, after the transition, are still coherent. Note that, the global observable behaviours are those which can be witnessed globally (introduced in Section 7.5). We formalise it because, during runtime, there is always a time lag due to asynchrony making the local specifications for the sender and the receiver inconsistent.

Specifying stateful specifications. This part is covered in Chapter 8, a full version of my paper (40), which contributes:

1. An introduction of an intuitive stateful specification, which is suitable for describing asynchronous stateful behaviours among multi-party sessions, and enriches (22) with set-based stateful operations (Sections 8.1 and 8.2 in Chapter 8).
2. An identification of a *semantic* issue when specifying asynchronous interactive behaviour combined with updatable states.
3. A formal analysis of the issue through *asynchronous trace semantics*. We reach several criteria for asynchronous verifiability of specifications (healthiness conditions (81)), including a decidable one which can express a rich set of specifications.
4. The properties of stateful specifications, which include the definition and analysis of the strongest asynchronous specification w.r.t a given synchronous one, and the theorems of safety satisfaction and the validation of stateful specifications.

1.3 Publications & Detailed Contributions of the Author

1. Kohei Honda, Aybek Mukhamedov, Gary Brown, **Tzu-Chun Chen**, Nobuko Yoshida: Scribbling Interactions with a Formal Foundation. ICDCIT 2011: 55-75. (83).

   Author's contribution: The Scribble language was first developed by Dr Kohei Honda and Dr Gary Brown. My contribution was part of the theoretical foundation for Scribble. This work was majorly carried out under the advice and the support of Dr Kohei Honda. The development of Scribble, which is not in the scope of this thesis, is ongoing (87).

   Corresponding Part: Chapter 9.

2. **Tzu-Chun Chen**, Laura Bocchi, Pierre-Malo Deniélou, Kohei Honda, Nobuko Yoshida: Asynchronous Distributed Monitoring for Multi-party Session Enforcement. TGC 2011: 25-45. (39).

Author's contribution: Parts of theories and ideas. The ideas and formulations of asynchronous linear session establishment, the permutation mechanism of specifications and global observables, paper's writing (mainly for the theory parts).

Corresponding Part: Chapters 5, 6, and 7.

3. **Tzu-Chun Chen**, Kohei Honda: Specifying Stateful Asynchronous Properties for Distributed Programs. CONCUR 2012: 209-224. (40).

   Author's contribution: The ideas, theories, writings, and proofs. This is a paper of my own.

   Corresponding Part: Chapter 8.

4. Laura Bocchi, **Tzu-chun Chen**, Romain Demangeon, Kohei Honda, and Nobuko Yoshida: Monitoring Networks through Multiparty Session Types. FORTE/FMOODS 2013. To appear (20).

   Author's contribution: Parts of theories and writings. The complete proofs of local/global safety, local/global transparency, and session fidelity.

   Corresponding Part: Chapters 5, 6, and 7.

1.4 Synopsis

The dissertation is divided into 9 chapters, covering the background and related works, the main theories, and the future directions.

Chapter 2 introduces the background of large-scale distributed systems, our leading example, and the asynchronous π-calculus and the multi-party session types, which are the theoretical foundations of this thesis.

Chapter 3 gives an overview of related works and the comparisons of our work with those related ones.

Chapter 4 gives the assumptions and defines the terminologies used in this thesis. This chapter provides an overall picture of the system and framework that we are interested in, and the assumptions supporting our formalisms and theories.

Chapter 5 introduces the capability-passing π-calculus. This chapter, although does not include any main theory, gives the basis for the design of governance specifications and the calculus of dynamic asynchronously monitoring.

Chapter 6 introduces the syntax and semantics of the session-type based global/local specifications. This chapter grounds the specifications used in the following chapters and introduces the consistency principles and several useful mechanisms for working with specifications, such as the projection and the permutation rules.

Chapter 7 introduces the dynamic asynchronously monitoring calculus on the base established in Chapter 5 and 6. Several main theorems, such as local/global safety, local/global transparency, and session fidelity, are established here.

Chapter 8 extends the specifications, introduced in Chapter 6, to stateful specifications. It addresses the semantic issue while the stateful specifications are applied to external monitors in asynchronous monitoring environments. Also, it proposes theories and several properties of satisfaction of the sequence of actions and the validation of stateful specifications.

Chapter 9 concludes the dissertation by summarising the main theory chapters (i.e. Chapters 5, 6, 7, and 8). It also focuses on important future work which can pave the way to design session-based policy languages useful in practice based on the fundamental theories developed in the main parts.

Appendix includes three parts: Part A contains the notes of Chapter 5. Part B contains all proofs of the theorems introduced in Chapter 7 and their auxiliary definitions and lemmas. Part C contains the proofs of results from Chapter 8 which are not proved directly there.

2

Background and Theoretical Foundations

2.1 Large-scale Distributed Systems

Tanenbaum and Steen (155) define a distributed system as "a collection of independent computers that appears to its users as a single coherent system."; while Leslie Lamport (12) says "you know you have a distributed system when the crash of a computer you've never heard of stops you from getting any work done". According to them, a distributed system is a collection of computing tasks that sometimes goes very wrong because of its complexity and the difficulties of its maintenance. In this dissertation, we are concerned with large-scale distributed systems from the point of view of their usage. From our perspective, the key property is:

> people residing in different domains want to conveniently and efficiently enjoy cooperation on tasks across domains.

Here "people" can be replaced by system components or any electronic devices. The reference to "different domains" implies that they may have different system policies, access control rules, and different verification mechanisms. The requirement of "conveniently and efficiently" implies that time and spatial limitations should be minimised. And "cooperation on tasks across domains" implies that if a user wants to access cooperative data in another domain, he or she should be able to do so through proper authentication. In a *large-scale* distributed system, participants may be geographically remote

and the system must be scalable enough to cater for heterogeneous remote components joining or leaving the system.

Jean Bacon carried out an extensive survey of large distributed systems and concluded that they are hard to maintain (14). The particular challenges that she identified include *concurrency*, *naming*, and *failure recovery*. The same issues are also prominent in the analysis of Anderson (12). This thesis, based on the π-calculus, takes concurrency of processes as the norm, uses names to denote endpoints, and equips primitives for name passing to control mobility. For failures (i.e. errors), the monitoring mechanism proposed in this thesis straightforwardly stops the ill-behaved process and drops it off. As the first step in governing asynchronous interactions with a session-based approach, this thesis does not cover the issue of a failure recovery mechanism.

2.2 Ocean Observatories Initiative (OOI)

The Ocean Observatories Initiative (OOI), the *model* of a large-scale distributed system that this thesis based on, is a major project funded by the US NSF. One of its goals is to realise the *cross-domain* inter-networks and inter-cooperations. The OOI's intention is not to produce a private cloud, unlike Google (70) and Amazon (8), it aims to provide a *public cloud* integrating more than one cloud to serve researchers and the public with persistent and interactive capabilities for observing the oceans over a period of more than 30 years (36, 129), and, at the same time, enjoys scalability and consistency. It therefore presents a general architecture which has all the attributes a large-scale distributed system should have. The governance theories in this thesis are mainly based on OOI's requirements and attributes.

With our ongoing cooperation, the governance theories proposed in this thesis are going to be adopted as the theoretical foundation for OOI's design. The key element is a comprehensive cyberinfrastructure (i.e. CI), whose design is based on loosely-coupled distributed services and agents throughout the OOI observatories, from seafloor instruments to on-shore research stations, communicating through a common messaging infrastructure. Most of the use case scenarios focus on distributed and structured conversations among endpoints which may be a thousand miles apart.

In conclusion, for OOI's system, OOI wishes to provide a communication and data transformation platform for ocean science, connecting observation, data collection, and

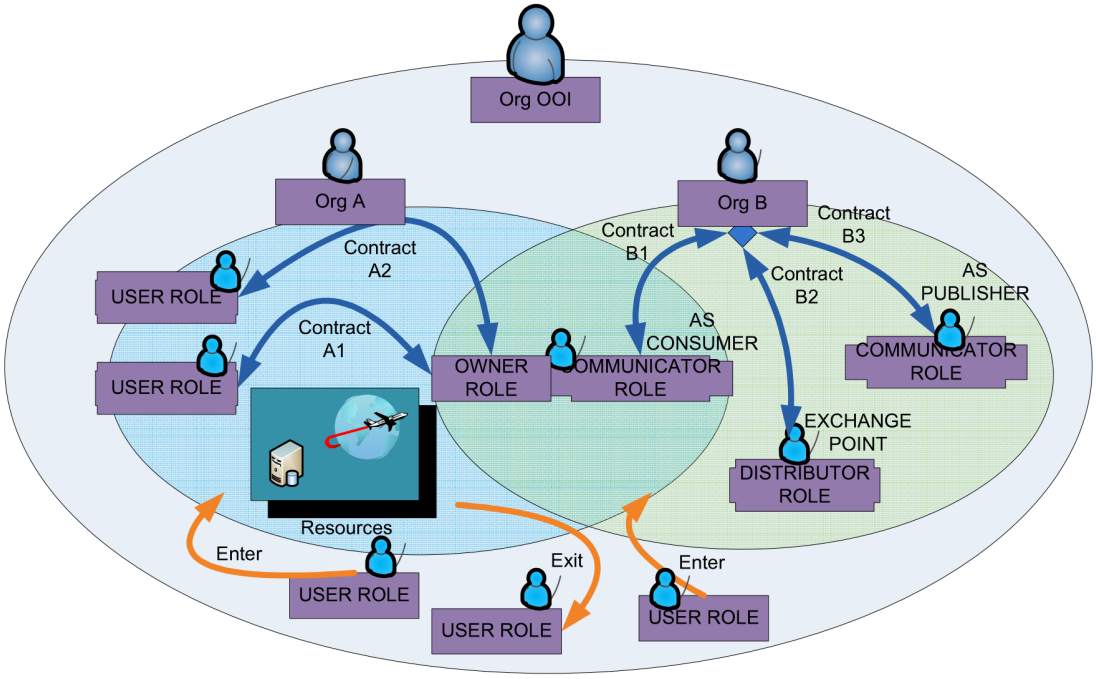

Figure 2.1: A Simplified Schematic Representation of Endpoints' Interactions

modelling under a mechanism, that includes a suitable governance system to guarantee stability and integrity. A traditional data-centric CI has a central data management system that ingests data and serves it to users on a query basis. This however is not sufficient to accomplish the range of tasks ocean scientists will engage in when the OOI is implemented. Instead, a highly distributed set of capabilities is required.

Figure 2.1 describes possible relationships among OOI organizations (Orgs). Each ellipse delineates an Org, corresponding informally to a community of participants. Orgs may be nested within, be disjoint from, or partially overlap with other Orgs. Org OOI is the root Org in that it defines the identities for the participants involving in and provides the rules to monitor and enforce contracts among its participants (41).

2.3 Theory Foundations

In this section, we give a brief overview of the two fundamental theoretical bases of this thesis: the asynchronous π-calculus (24, 84) and session types (30, 69, 85, 86, 157), which structure communications. Since the asynchronous π-calculus is built upon the π-calculus (117, 118), we start from introducing the π-calculus to give readers an

overview. The π-calculus is a process algebra for modeling communications. There are many formalisms for describing concurrent communications besides the π-calculus. They include the actor model (76), CSP (communicating sequential processes) (79, 80), CCS (communication and concurrency system) (115), and the Ambient Calculus (33). This thesis is based on the asynchronous π-calculus for two main reasons:

1. Like the π-calculus, it has the virtue providing a neat and sufficient expressive model for identifying and formalising interactive behaviours among participants in distributed systems. This virtue is elegantly captured in its syntax and semantics, which will be reviewed below.

2. It identifies concurrent and asynchronous behaviours, and mobility of processes through message passing with a simple but expressive syntax and semantics.

We also make substantial use of the theory of *session types* (22, 30, 31, 57, 69, 86, 89). Session types, the types typing conversations, are simple and expressive formal languages for describing protocols. When we use a session type to define a session, although the interactive behaviours in an individual session may be sequential, *different sessions can interleave*, which makes the overall interactions behave *concurrently*. Thus the attribute of concurrency of processes in distributed systems is captured in our theory. Since the focus of this thesis is to establish a foundation for *governing* processes's interactions in large-scale distributed systems, applying the ideas of protocols, which define and specify the behaviours of session participants who agree on it, is thus practical, expressive, and effective.

2.3.1 The π-calculus

As noted by (134), the π-calculus (117, 118) is a core calculus of message-based concurrency. It was inspired by Nielsen and Engberg (59), and formulated and introduced by Robin Milner, Joachim Parrow and David Walker (118). There are many studies about the π-calculus, including the works of (82, 116, 117, 124). The defining feature of the π-calculus is its ability to model interactive behaviours among participants by message passing. As three main attributes of activities in large-scale distributed systems are: (I) interactions, including sending-receiving pair-wise communications via passing messages among processes, are the norm, and (II) the behaviours of those interactions

are majorly concurrent, and (III) the processes can be mobile in the network. The π-calculus, with its conciseness and expressiveness for concurrency and name passing, provides for this thesis an ideal basis.

2.3.1.1 The Syntax and Semantics

Here we briefly introduce the action labels and the grammar of the π-calculus.

Action Prefixes

The *action prefixes* α of the π-calculus includes both observable and internal (τ) actions. An action prefix represents either sending or receiving a message (a name), or making a silent transition. The syntax is

$$\begin{array}{rcll} \alpha & ::= & x(y) & \text{receiving (input) of name } y \text{ via channel } x \\ & | & \overline{x}\langle y\rangle & \text{sending (output) of name } y \text{ via channel } x \\ & | & \tau & \text{internal (silent) action} \end{array}$$

Processes

The set of the π-calculus processes, P, Q, ... , is defined by the grammar

$$\text{Process } P ::= \textstyle\sum_{i\in I} \alpha_i.P_i \mid P_1|P_2 \mid \nu x P \mid !P,$$

where I is any finite indexing set. The process $\sum_{i\in I} \alpha_i.P_i$ is called summation or sum. It represents *guarded choice*. Generally speaking, P_i is guarded by α_i, since the action represented by α_i must occur before P_i becomes active. $\mathbf{0}$ (inactive) is used to represent the empty sum, $\alpha.P$ (prefix) is used to represent sum on one element only, and $P+Q$ is for the binary sum. The symbols | and ! are respectively the *parallel* and the *replication* operator. The *restriction* νx and the input $y(x)$ both bind the name x, i.e. in the processes $\nu x P$ and $y(x).P$, the occurrences of x in P are considered bound with the usual rules of scoping. On the other hand, x is free on the output $\overline{y}\langle x\rangle$ action. The set of free names of P, i.e. those names which do not occur in the scope of any binder, is denoted by $\mathsf{fn}(P)$. The set of bound names is denoted by $\mathsf{bn}(P)$, and $\mathsf{n}(P)$ is used to denote all the names which occur in P. The renaming (or substitution) $P\langle y/x\rangle$ is defined as the result of replacing all free occurrences of x in P by y.

Process Contexts

The set of process contexts C is given by the syntax:

$$C ::= [\,] \mid \alpha.C + M \mid \nu x C \mid C|P \mid P|C \mid !C.$$

$C[Q]$ denotes the result of filling the hole $[\,]$ in the context C with the process Q. The elementary contexts are $\alpha.[\,] + M$, $\nu x[\,]$, $[\,] \mid P$, $P \mid [\,]$, and $![\,]$.

Process Congruence

Let $\cong$ be an equivalence relation over the set of all the π processes. The $\cong$ is said to be a process congruence if it is preserved by all elementary contexts; that is, if $P \cong Q$, then

$$\begin{aligned} \alpha.P + M &\cong \alpha.Q + M \\ \nu x P &\cong \nu x Q \\ P \mid R &\cong Q \mid R \\ R \mid P &\cong R \mid Q \\ !P &\cong !Q \end{aligned}$$

Structural Congruence

Two process expressions P and Q in the π-calculus are structurally congruent, written $P \equiv Q$, if we can transform one into the other by using the following equations (in either direction)

- Change of bound names (*alpha-conversion*)
- Reordering of terms in a summation
- $P \mid 0 \equiv P$, $P \mid Q \equiv Q \mid P$, $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$
- $\nu x(P \mid Q) \equiv P \mid \nu x Q$ if $x \notin \mathsf{fn}(P)$
- $\nu x 0 \equiv 0$, $\nu x y P \equiv \nu y x P$
- $!P \equiv P \mid !P$

Reduction

The reduction rules are define below:

$$\text{TAU}: \quad \tau.P + M \longrightarrow P$$

$$\text{REACT}: \quad (x(y).P + M) \mid (\overline{x}\langle z\rangle.Q + N) \longrightarrow P\{z/y\} \mid Q$$

$$\text{PAR}: \quad \frac{P \longrightarrow P'}{P|Q \longrightarrow P'|Q}$$

$$\text{RES}: \quad \frac{P \longrightarrow P'}{\nu x P \longrightarrow \nu x P'}$$

$$\text{STRUCT}: \quad \frac{P \longrightarrow P'}{Q \longrightarrow Q'} \text{ if } P \equiv Q \text{ and } P' \equiv Q'$$

We use the following example to illustrate reduction rules. A reduction is a silent action (τ action). This example is extracted from Robin Milner's *Communicating and Mobile Systems : The π-calculus* (117). Let

$$P = \nu z((\overline{x}\langle y\rangle + z(w).\overline{w}\langle y\rangle) \mid x(u).\overline{u}\langle v\rangle \mid \overline{x}\langle z\rangle).$$

x, y, and v are the free names in P. A pair of positive and negative actions using the same channel, such as $\overline{x}\langle y\rangle$ and $x(u)$ are complementary. If both are unguarded and not in the same summation (i.e. not alternatives to each other) then they constitute a *redex*: this redex constitutes a reduction $P \longrightarrow P'$, which invokes a substitution-here $\{y/u\}$. In the example, P has two redexes; the pair $\overline{x}\langle y\rangle$, $x(u)$ and the pair $x(u)$, $\overline{x}\langle z\rangle$. By the rule REACT, there are two possible reductions $P \longrightarrow P_1$, where $P_1 = \nu z(0 \mid \overline{y}\langle v\rangle \mid \overline{x}\langle z\rangle)$, or $P \longrightarrow P_2$, where $P_2 = \nu z((\overline{x}\langle y\rangle + z(w).\overline{w}\langle y\rangle) \mid \overline{z}\langle v\rangle \mid 0)$. There is no further reduction in P_1, but by rule REACT there is one in P_2:

$$P_2 \longrightarrow P_3, \text{ where } P_3 = \nu z(\overline{v}\langle y\rangle \mid 0 \mid 0)$$

This example also shows the non-determinism of concurrent processes.

Labelled Transition System

The labelled transition systme (LTS) is defined as follows:

$$\text{Actions } \ell ::= x(y) \mid \overline{x}\langle y\rangle \mid \overline{x}(y)$$

$$\text{I-SUM} \quad \Sigma_i \alpha_i.P_i \xrightarrow{x(z)} P_j\{z/y\} \qquad \alpha_j = x(y)$$

$$\text{O}/\tau\text{-SUM} \quad \Sigma_i \alpha_i.P_i \xrightarrow{\alpha_j} P_j \qquad \alpha_j = \overline{x}\langle y\rangle \text{ or } \alpha_j = \tau$$

$$\text{OPEN} \quad \frac{P \xrightarrow{\overline{x}\langle y\rangle} P'}{(\nu y)P \xrightarrow{\overline{x}(y)} P'}$$

$$\text{RES} \quad \frac{P \xrightarrow{\ell} P'}{(\nu y)P \xrightarrow{\ell} (\nu y)P'} \qquad y \notin \mathsf{n}(\ell)$$

$$\text{PAR} \quad \frac{P \xrightarrow{\ell} P'}{P \mid Q \xrightarrow{\ell} P' \mid Q} \qquad \mathsf{fn}(Q) \cap \mathsf{bn}(\ell) = \emptyset$$

$$\text{COM} \quad \frac{P \xrightarrow{\overline{x}\langle y\rangle} P' \;\; Q \xrightarrow{x(y)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$\text{CLOSE} \quad \frac{P \xrightarrow{\overline{x}(y)} P' \;\; Q \xrightarrow{x(y)} Q'}{P \mid Q \xrightarrow{\tau} (\nu y)(P' \mid Q')} \qquad y \notin \mathsf{fn}(Q)$$

$$\text{REP} \quad \frac{P \mid !P \xrightarrow{\ell} P'}{!P \xrightarrow{\ell} P'}$$

$$\text{CONG} \quad \frac{P_1 \equiv P_2 \;\; P_2 \xrightarrow{\ell} Q_2 \;\; Q_2 \equiv Q_1}{P_1 \xrightarrow{\ell} Q_1}$$

Note that $\mathsf{n}(\ell)$ means the collection of free and bound names occuring in ℓ. $\mathsf{n}(\ell)$ does not include the channel name. For example, $\mathsf{n}(\overline{x}\langle y\rangle) = \mathsf{n}(x(y)) = y$. $\mathsf{bn}(\ell)$ means the name bound by channel, e.g. $\mathsf{bn}(x(y)) = \mathsf{bn}(\overline{x}(y)) = y$, but $\mathsf{bn}(\overline{x}\langle y\rangle) = \mathsf{bn}(\tau) = \emptyset$. And, on the contrary, $\mathsf{fn}(\ell)$ means the name free to channel, e.g. $\mathsf{fn}(\overline{x}\langle y\rangle) = y$, $\mathsf{fn}(x(y)) = \mathsf{fn}(\overline{x}(y)) = \mathsf{fn}(\tau) = \emptyset$. Rules I-SUM and O/$\tau$-SUM are straightforward: When the action of an input (resp. output) process fires, the process executes the input (resp. output) action; while τ action is for reduction of processes. The only action different from the prefix (i.e. α) is $\overline{x}(y)$, which means a bound output. It is introduced to model scope extrusion, which is corresponding to transition rule OPEN. Scope extrusion is

outputting a secret, say a private name y (through bound output $\overline{x}(y)$). If there is no corresponding receiver to get this private name (see rule CLOSE), then this private name becomes non-private. Rule RES says, when the action has nothing to do with the private names possessing by process $(\nu y)P$, after the fire of this action, $(\nu y)P'$ still has this name as private. If the channel of ℓ is exactly the private name y, it means the message is delivered internally. Rule PAR says the action will not affect the concurrent process Q if $\mathsf{fn}(Q) \cap \mathsf{bn}(\ell) = \emptyset$, e.g. it will not result in a communication, as rule COM defines. Rule COM corresponds to the reduction rule REACT: When there synchronously exist an output $\overline{x}\langle y\rangle$ and an input $x(y)$, a communication happens. While rule CLOSE, which seems similar to COM, is a dual to rule OPEN. It says when the sender sends a secret (by sending a private name via bound output $\overline{x}(y)$) and at the same time, there is a receiver ready to input this secret, then this secret is shared only by the sender and the receiver, denoted by $(\nu y)(P' \mid Q')$, saying the name is private in the scope of $(P' \mid Q')$.

2.3.1.2 Mobility

As Milner's work (117) states, in the π-calculus, *link* is used as the connectivity for describing *mobility*. The movement of a process can be represented entirely by the movement of its links, which is called as *link mobility*. According to Milner's observation, what distinguishes the π-calculus from earlier process calculi, in particular to Calculus of Communicating Systems (CCS) (115) and Tony Hoare's similar work on Communicating Sequential Processes (CSP) (79, 80), is the capability to change the links of a network of processes.

In the π-calculus, that is the *name-passing* makes mobility possible:

$$\begin{array}{rl} \overline{x}\langle y\rangle.P \mid x(z).\overline{z}\langle 10\rangle.Q \longrightarrow P \mid \overline{y}\langle 10\rangle.Q\{y/z\} & \text{the } \pi\text{-calculus} \\ \overline{x}\langle 10\rangle.P \mid x(z).\overline{y}\langle z\rangle.Q \longrightarrow P \mid \overline{y}\langle z\rangle\{10/z\}.Q\{10/z\} = \overline{y}\langle 10\rangle.Q\{10/z\} & \text{the CCS}_{vp} \end{array}$$

In the π-calculus, a name y is passed as a message, from process $\overline{x}\langle y\rangle$ via channel x to another concurrent process $x(z).\overline{z}\langle 10\rangle.Q$. As reduction happens (i.e. $x(z).\overline{z}\langle 10\rangle.Q$ receives y), y becomes a channel, $\overline{y}\langle 10\rangle$, for sending out value 10. While in the CCS$_{vp}$ (i.e. value-passing CCS), a name cannot be passed, but only a value can be passed. Thus in the π-calculus, when a name is passed as a message, it can later become a channel. This simple idea realises the mobility through the change of links, which dynamically

change the computing configurations. Palamidessi in (133) analysed that link mobility is a very powerful feature for coordination of distributed activities, since it allows two remote processes, originally not directly connected, to establish a direct communication link and to take decision together via synchronous exchange of information along that link. Palamidessi proved that the link mobility makes the π-calculus strictly more powerful than CCS_{vp}.

2.3.2 The Asynchronous π-calculus

Asynchrony is inherent in distributed systems because it takes time for a message to travel from one system component to another. The asynchronous π, proposed by Honda and Tokoro in 1991's (84) and, independently by Boudol in 1992's work (24), contains no explicit operators for choice and output-prefixing. Due to the lack of output prefix, an output action is non-blocking and can only be written "in parallel" with other activities. With the π-calculus introduced above, here we address the main different grammars for the asynchronous π-calculus.

Asynchronous Processes

The set of the asynchronous π-calculus processes, P, Q, ... , is defined by

$$\text{Process } P ::= \overline{x}\langle y\rangle \mid x(y).P \mid \textstyle\sum_{i\in I} P_i \mid P_1|P_2 \mid \nu x P \mid\ !P,$$

The main difference is, in the asynchronous π-calculus, there is no output-prefix guarding a process. The output is unattached to the process producing it but exists in parallel with it. This design allows the sender to output without waiting for the synchronisation of the receiver side.

Labelled Transition Sytem

The main rules different from the LTS rules of the π-calculus are listed below. Let $\overline{x}y ::= \overline{x}\langle y\rangle \mid \overline{x}(y)$:

$$\text{OUT} \qquad \overline{x}y \xrightarrow{\overline{x}y} \mathbf{0}$$

$$\text{COM} \quad \overline{x}y, x(z).P \xrightarrow{\tau} P\{y/z\}$$

Rule OUT states that $\overline{x}y$ is able to be outputted at anytime and is independent to other processes at one location. In rule COM, "," is used to separate two processes inside one

location i.e. $\overline{x}y$ and $x(z).P$ are both at one location. The calculus proposed in this thesis uses different notation for "," and "|". We use | for "," to denote that a local process can consist of multi-threads in parallel; while use ‖ for "|" to represent that the local processes existing in the network in parallel.

The attributes and benefits of the asynchronous π-calculus can be observed as making the communication between a sender and a receiver becomes asynchronous through having the sender to be able to fire its output at every possible execution time.

2.3.2.1 Synchrony v.s. Asynchrony

In concurrent and distributed computations, communications are dominant. Communication is a process for delivering messages, exchanging opinions, negotiation, and achieving agreements. There are many kinds of communication models, for example, synchronous, asynchronous, one-to-one, or one-to-many communication models. Two main models are discussed: The synchronous π-calculus, denoted by π, and the asynchronous π-calculus, denoted by π_a. We study the distinction between the synchronous and the asynchronous π-calculus with Palamidessi's work (133). Note that the π-calculus itself is a synchronous paradigm which contains an "asynchronous" fragment (Boudol, 1992; Honda and Tokoro, 1991).

In the π-calculus, we write $P = \overline{x}\langle y\rangle.P^{'}$ to represent a process P that performs an output of name y on channel x and continues as $P^{'}$, and we write $Q = x(z).Q^{'}$ to represent a process Q that performs an input on channel x and continues as $Q^{'}\{y/z\}$ after replacing every z in $Q^{'}$ with y. By using these operators, the synchronisation of P and Q on x is enforced: Only when the communication along x takes place, these two processes can proceed. On the contrary, in the asynchronous π-calculus, we write a process that performs an output in the form $P = \overline{x}\langle y\rangle \mid P^{'}$, where $\overline{x}\langle y\rangle$ is an output to send y through channel x and | is the operator representing parallel composition. Since they are in parallel, the output process does not have to wait until a reception with corresponding primitive, say $x(z)$, appears. In other words, the output action does not suspend the continuous process, i.e. P', and there is no crucial notion of continuation point. $\overline{x}\langle y\rangle \mid P^{'}$ is a process which performs an output on x at some unspecified moment and of the handshaking between $\overline{x}\langle y\rangle$ and $x(z).Q^{'}$ as the moment in which the message is received by a reception. The reception enables the continuation point $Q^{'}\{y/z\}$ in the receiver, but it does not cause any resumption of activity in the sender.

In conclusion, a synchronous communication is usually understood as the simultaneous exchange of information between the partners, for example, the telephone. In contrast, in an asynchronous communication, the action of sending a message and the action of receiving it usually take place at different times, for example, the e-mail. (133) has proved that there does not exist any uniform, fully distributed translation from the π-calculus into the asynchronous π-calculus, up to any "reasonable" notion of equivalence. In other words, the asynchronous π, π_a, is not as expressive as the full π-calculus.

However, the communication model of π_a is powerful enough to simulate output-prefixing, as shown by Honda and Tokoro 1991 and by Boudol 1992, and input-guarded choice, as shown by Nestmann and Pierce 2000's work (128).

2.3.3 Session Types

A session type is a type for structuring a conversation, or called a "session", which consists of more than one *related interaction* and multiple participant. It forms and specifies a unit of series of interactions which generally involve multiple participants, called endpoints in this thesis. The interactions can be sequential interactions, parallel interactions, or sub-interactions (which are formally called sub-sessions in (49)). A session type is a protocol-like specification with formal shape.

Traditionally, viewing a computation as a sequence of actions gives a basis of the principles for sequential programming. Based on this viewpoint, sessions are done by RPC (remote procedure call). Every RPC is viewed as a single pair of interactions that starts with a request from a client to a server and ends with the response from the server to the client. The main difference between the concept of RPC and the one of sessions is that a session describes a structured flow of interactions which may include one or more, possibly an unbounded number of, interactions. For example, creating a new bank account is naturally abstracted as a scenario involving a sequence of interactions contacting with other services such as authentication components. From the viewpoint of RPC, both at the programming level and at runtime, such a sequence of interactions are not treated as forming a meaningful scenario as a whole since each call-return constitutes a single, isolated session. Figure 2.2 shows these two concepts as they both capture a common sequence of interactions but result in different representations of the behaviours. For the 4 interactions among Alice and Bob, they can be viewed

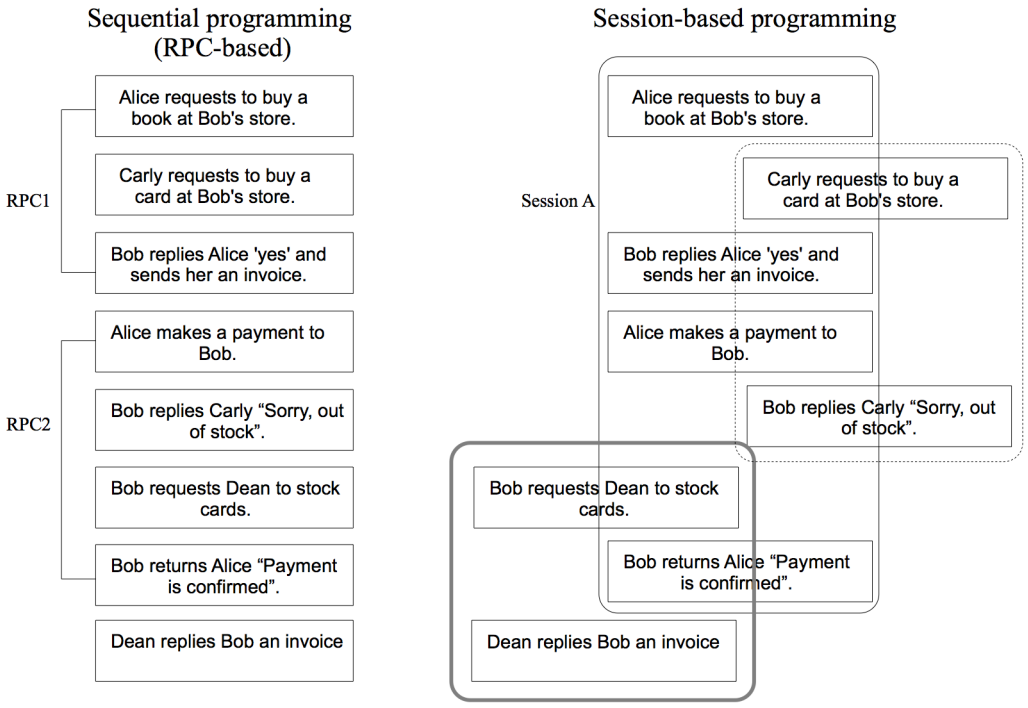

Figure 2.2: The Difference between Sequential and Session-based Abstractions of Interactions

as two independent RPCs (with two different RPC identities) or can be viewed as one session. The advantage of viewing these 4 interactions as one session is that it makes the relationships between endpoints trackable. For example, if some issue happens at the second RPC interaction, which is however caused by the first RPC interaction, without understanding that they are in fact related, the system cannot track through these interactions and resolve the problem. Viewing a session as a unit of executions makes it possible to clarify the patterns of interactive behaviours as a semantic entity.

2.3.3.1 Multi-party Session Types

The works of multi-party session types (*MPST*s) (32, 86) present a formal approach to the local validation of globally specified protocols, which assure that the session names are used linearly (i.e. linearity) and the interactions are deadlock-free in a single session (i.e. progress). Overall, session types ensure communication safety, protocol fidelity

and progress. Here we introduce its grammars of global and local types, and use one example in (86) to illustrate the use of session types.

The Syntax of Global and Local Types

In Figure 2.3, U is for *value types* which denote interaction message types. $T@p$ is a located type, representing a local type T assigned to participant p. S is for standard value types, such as $\mathsf{bool}, \mathsf{int}, \mathsf{string}$, etc., and the types for channels. $G, G', ...$, range over global types, $T, T', ...$, range over local types, and $p_1, p_2, ...$ range over session participants, and k is the variable for a channel. The rule of values for G says participant p_1 should send a message with type U to p_2 through channel k, then both participants should continue interactions guided by G'. From the local view point, participant p_1 has a corresponding local type, $k!\langle U\rangle; T'$, to guide it to send a message of type U through channel k, and continues the behaviour w.r.t. T'; while participant p_2 has a corresponding local type, $k?\langle U\rangle; T'$, to guide it to receive a message of type U through channel k, and continues the behaviour w.r.t. T'. The rule of branching for G says participants p_1 and p_2 should use channel k to interact through selecting a label from the branching pool (i.e. label pool). From the local viewpoint, participant p_1 has a corresponding local rule, called selection, to guide it to select a label, say l_j, and continues the actions specified by T_j; while participant p_2 has a corresponding local rule, called branching, to guide it to receive possible labels which may be chosen by the sender, and continues the actions specified by T_k if the label k has been chosen. The rule of recursion for global type, $\mu \mathsf{t}.G$, is the type for defining a recursion, where G is the recursion body. When we project it on the local endpoints, we have the corresponding recursion rule, $\mu \mathsf{t}.T$, where t is a type variable for either G or T. It is used as a recursion point when it is involved in a recursion. end is the termination of interactions. In summary, a global type G layouts the interactions among session participants globally. When we project a global type to all session participants, a set of local types are generated to guide the interactive behaviours of endpoints.

Classic Example

The following example shows a scenario that a session is established for three participants, *Buyer*, *Broker*, and *Seller*, in which *Buyer* sends a request to *Broker*, and

$$
\begin{array}{rcl}
U & ::= & S \mid T@p \quad \text{Value} \\
S & ::= & \mathsf{bool} \mid \ldots \mid \langle G \rangle \quad \text{Sort}
\end{array}
$$

$$
\begin{array}{rcll}
G & ::= & p_1 \to p_2 : k\langle U \rangle.G' & \text{values} \\
 & \mid & p_1 \to p_2 : k\{l_i : G_i\}_{i \in I} & \text{branching} \\
 & \mid & G \mid G' & \text{parallel} \\
 & \mid & \mu \mathsf{t}.G & \text{recursive} \\
 & \mid & \mathsf{t} & \text{variable} \\
 & \mid & \mathsf{end} & \text{end}
\end{array}
\qquad
\begin{array}{rcll}
T & ::= & k!\langle U \rangle; T & \text{send} \\
 & \mid & k?\langle U \rangle; T & \text{receive} \\
 & \mid & k \oplus \{l_i : T_i\}_{i \in I} & \text{selection} \\
 & \mid & k\&\{l_i : T_i\}_{i \in I} & \text{branching} \\
 & \mid & \mu \mathsf{t}.T \mid \mathsf{t} \mid \mathsf{end} &
\end{array}
$$

Figure 2.3: The Grammar of Global (G) and Local (T) Types

Broker classifies this request should belong to either catalog buy or $\mathsf{service}$. Then *Broker* informs *Seller* the catalog (i.e. branch label) she chooses. If label buy is chosen, both *Broker* and *Seller*'s interactions should follow G_{buy}'s guidance, otherwise they should follow G_{service}'s guidance.

We use global type G to formally represent this scenario:

$$
\begin{array}{rcl}
G & = & \mathit{Buyer} \to \mathit{Broker} : a\langle \mathsf{string} \rangle. \\
 & & \mathit{Broker} \to \mathit{Seller} : b\{\mathsf{buy} : G_{\mathsf{buy}}.\mathsf{end}, \mathsf{service} : G_{\mathsf{service}}.\mathsf{end}\}
\end{array}
$$

Note that, G indicates that channel a is used for *Buyer* to communicate with *Broker*, who is responsible for delivering the request of type string, from *Buyer* to *Seller*; while the communication between *Broker* and *Seller* is done through channel b.

For the endpoints, they respectively have the following local types:

$$
\begin{array}{rcl}
T_{\mathit{Buyer}} & = & a!\langle \mathsf{string} \rangle; \mathsf{end} \\
T_{\mathit{Broker}} & = & a?\langle \mathsf{string} \rangle; b \oplus \{\mathsf{buy} : T^{\mathit{Broker}}_{\mathsf{buy}}; \mathsf{end}, \mathsf{service} : T^{\mathit{Broker}}_{\mathsf{service}}; \mathsf{end}\} \\
T_{\mathit{Seller}} & = & b\&\{\mathsf{buy} : T^{\mathit{Seller}}_{\mathsf{buy}}; \mathsf{end}, , \mathsf{service} : T^{\mathit{Seller}}_{\mathsf{service}}; \mathsf{end}\}
\end{array}
$$

$T^{\mathit{Broker}}_{\mathsf{buy}}$ is the local type for participant *Broker* when label buy is selected, while $T^{\mathit{Broker}}_{\mathsf{service}}$ is for this participant when label $\mathsf{service}$ is selected. Similary, $T^{\mathit{Seller}}_{\mathsf{buy}}$ and $T^{\mathit{Seller}}_{\mathsf{service}}$ are the local types for participant *Seller* when label buy and label $\mathsf{service}$ are selected, respectively. This example shows that session types structure multiple interactions to an organised interaction flow and guide endpoints to behave well as what global scenario expects.

$$
\begin{array}{lcll}
L & ::= & \{p, p', ...\} & \text{Location} \\
S & ::= & \mathsf{bool} \mid ... \mid \langle G \rangle & \text{Sort}
\end{array}
$$

$$
\begin{array}{lcll}
G & ::= & p_1 \to p_2 : k\langle \tilde{v} : \tilde{S} \rangle \{A\}.G' & \text{values} \\
 & \mid & p_1 \to p_2 : k\{\{A_i\} l_i : G_i\}_{i \in I} & \text{branching} \\
 & \mid & G \mid G' & \text{parallel} \\
 & \mid & \mu \mathsf{t} \langle \tilde{e} \rangle (..., \tilde{v_i} : \tilde{S_i} @ L_i) \{A\}.G & \text{recursive} \\
 & \mid & \mathsf{t} \langle \tilde{e} \rangle & \text{variable} \\
 & \mid & \mathsf{end} & \text{end}
\end{array}
\qquad
\begin{array}{lcll}
T & ::= & k!\langle \tilde{v} : \tilde{S} \rangle \{A\}; T & \text{send} \\
 & \mid & k?\langle \tilde{v} : \tilde{S} \rangle \{A\}; T & \text{receive} \\
 & \mid & k \oplus \{\{A_i\} l_i : T_i\}_{i \in I} & \text{selection} \\
 & \mid & k\&\{\{A_i\} l_i : T_i\}_{i \in I} & \text{branching} \\
 & \mid & \mu \mathsf{t} \langle \tilde{e} \rangle (\tilde{v} : \tilde{S}) \{A\}.T \mid \mathsf{t} \mid \mathsf{end} &
\end{array}
$$

Figure 2.4: The Syntax of Global (G) and Local (T) Assertions

2.3.3.2 Multi-party Session Assertions

Multi-party session assertions (*MPSA*s) (22) further allow, by extending *MPST*s with logical formulae, fine-grained specifications of interactive behaviour including message contents, runtime choices of conversation paths and recursion invariants. By projecting a *global assertion* (i.e. global protocol specification) onto endpoints, we get *endpoint assertions* (i.e. local protocol specifications), each of which specifies endpoint in the protocol. Thus the local validations which automatically ensure global correctness are obtained.

The Grammar of Global and Local Assertions

Figure 2.4 shows the syntax of *MPSA*s, which extend *MPST*s with logical formulae. The syntax of *MPSA*s is similar to the one of *MPST*s, except that the interaction predicate A is introduced at every interaction to guard interactions. Note that, $T@p$ in *MPST*s is replaced by $v : S@L$ (particularly defined in recursion for global types) with definition of locations $L ::= \{p, p', ...\}$. Also note that $\tilde{v}$ is defined in rule of values, send, receive, and recursion because $\tilde{v}$ should be specified in interaction predicate A to bind A. As for the type of global recursion, since it defines the recursion variables known by all endpoints involving in the recursion, in $(..., \tilde{v_i} : \tilde{S_i}@L_i, ...)$, $\tilde{v_i} : \tilde{S_i}@L_i$ indicates the types of contents for a particular location L_i; while for the type of local recursion, since it has been located, only $\tilde{v} : \tilde{S}$ is enough.

3

Related Works and Comparisons

To represent our focus without confusion, here we only mention the most related works having significant impact to the subjects that we are interested in. As a brief summary of comparisons, the following two points are noted here:

1. In the literature, the works of dynamic monitoring mostly rely on inlined monitors. In other words, in these works, they monitored the asynchronous interactions among endpoints through embedding monitoring code besides the variables and procedures which should be protected; however, as an external monitor is adopted for economic and scalable settings, we shall consider the asynchronous interactions between an endpoint and its corresponding monitor.

2. In the literature, most works of runtime monitoring do not provide the theoretical foundation to state and prove that their mechanisms enjoy the properties of safety and transparency, which are necessary for governing large-scale distributed systems. This thesis gives both. To cater for the asynchronous monitoring, we provide the semantics and the permutation mechanism of specifications for monitoring, which determines the endpoint behaviours based on these specifications, to capture the effects of actions whose order may not be preserved due to concurrency and asynchrony.

3.1 Specification and Verification for Concurrent Programs

Specifications are the rules for statically regulating endpoint programs and system structures or dynamically verifying runtime behaviours of processes and systems. In Alfred V. Aho et al 's book (5), a specification is as an articulation of the request of a compiler. In Pierce's book (135), a specification is defined by a type inference system. In general, a specification is a concise description which formally specifies the intended computation outcomes and makes verification of programming code or runtime processes possible. A proper specification should make verification effective. When the verification follows the semantics of a specification, it can at least judge an invalid code or process as incorrect, and can at most judge a valid code or process as correct.

There are two aspects of formal verification: One is to statically inspect the structure of programming code, the other is to dynamically check the behaviours of processes. Leslie Lamport's opening in (101) says "a large body of research on formal verifications can be characterised as state-based or assertional". A state-based verification is an automata-like approach, where the "state" means the "status" of a program or a process. An assertional verification is an approach with proof system and the construction of invariants, which are logical formulae whose value never change (100, 131).

The following history of specification and verification for concurrent programs is a summary from Lamport's note (101), which reviewed the works having significant contributions and advocates specifying programs with mathematics (e.g. the proposal of TLA (temporal logic of actions)). In 1967, Floyd in (64) introduced the modern concept of correctness for sequential programs. In every program, there are control points, called annotations, attached with assertions, which are written in logical formulae. Partial correctness is proved by verifying if the control point is annotated "*true*" whenever control is at the point. A formula in Hoare's logic, proposed in 1969's work (77), has the form $\{P\}S\{Q\}$, denoting that if the assertion P is true before initiation of program S, then the assertion Q will be true upon S's termination. The state-based approach viewed a program as a state transformer, which can be verified by checking whether every control point satisfies the attached assertions.

Ashcroft in 1975 (13) adopted the state-based approach to concurrent programs, where concurrency is expressed by fork and join operations. In concurrent programming, a control can be at multiple control points at the same time: At every moment,

the annotation of a control is not unique, but rather as a collection of annotations of controls representing that the assertions are true. This collection is viewed as *invariance*, which should be true after every execution. The idea of invariance gives a fundamental principle to specify concurrent programs: Instead of only considering what is true before and after executing the program, one must consider what remains true throughout the execution.

In 1976, Owicki and Gries (131) attempted to reason about concurrent programs by generalising Hoare's method with the following additional proof rule:

$$\frac{\{P_1\}S_1\{Q_1\}, ..., \{P_n\}S_n\{Q_n\}}{\{P_1 \cap ... \cap P_n\}\ \textbf{cobegin}\ S_1 \parallel ... \parallel S_n\ \textbf{coend}\ \{Q_1 \cap ... \cap Q_n\}}$$

provided $\{P_1\}S_1\{Q_1\}, ..., \{P_n\}S_n\{Q_n\}$ are interference-free.

Unlike Ashcroft's method, the assertions in the Owicki-Gries method do not mention the control state but only introduce auxiliary variables to capture the control information. Ashcroft's invariance is actually hidden inside the machinery of the Owicki-Gries method: This proof rule is as a general summary of valid concurrent programs without specifying how to share variables among concurrent programs to achieve invariance. To achieve $\{P_1 \cap ... \cap P_n\}$ **cobegin** $S_1 \parallel ... \parallel S_n$ **coend** $\{Q_1 \cap ... \cap Q_n\}$, P_i and P_j $(i \neq j)$, P_i and Q_j $(i \neq j)$ should have some hidden relationships which are not fully addressed.

Lamport in (99) indicates that a module in a concurrent program must be specified in terms of its behaviour, rather than the values it returns because, in a concurrent program, input and output actions are no more sequentially executed in one process but could be called by different processes. With the reality of concurrency, temporal logic is a useful tool to reason about the behaviour of concurrent programs (132, 137). In (99), Lamport introduced temporal assertions to make the specifications simpler and easier to understand. In (27), distributed temporal logic (DTL) was proposed for formalising properties of concurrent, communicating agents. It is a modal temporal logic with modalities referring to events, which are mapped to corresponding actions. (27) also introduced communication points to realise the asynchrony between a sender and a receiver through two phases of synchronisations: Synchronising the sender and a shared channel (i.e. sending a content to the communicating channel), and then synchronising the channel and the receiver (i.e. receiving the content from the communicating channel).

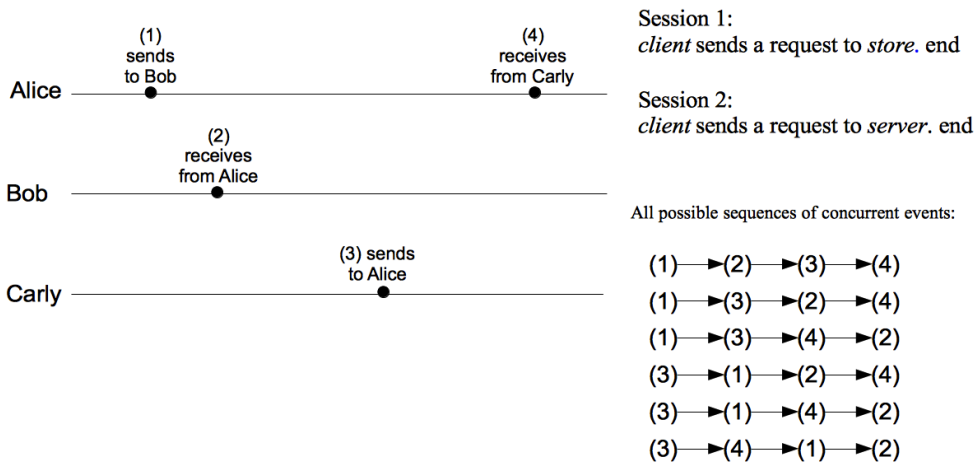

Figure 3.1: The Different Approaches of the Representation of the Relationships of Events

To compare the session-type-based approach and the temporal logic-based one, we use Figure 3.1 to illusrate that the session-type-based approach gives a cleaner and more uniform specification for verification. Assume there are three principals, Alice, Bob, and Carly in the system. Each of them has a unique identity. Assume Alice has a communication with Bob: Alice orders a book from Bob. Here Alice plays role *client* and Bob plays role *store*. Also, assume Alice has another communication with Carly: Carly sends a task request to Alice. Here Alice plays role *server* and Carly plays role *client*. For simplicity, here we only show the one-way events (i.e. no corresponding responses). In the figure, a horizontal solid line is the life time of a principal, and a black spot is an event. These two interactions are concurrent. As we apply the temporal logic-based approach, we have at least 6 different possible temporal assertions to describe these two concurrent interactions. As we apply the session-type-based approach, since unrelated sessions, e.g. the interaction of Alice and Bob and the one of Alice and Carly are unrelated, can interleave, only two session types (i.e. Session 1 and Session 2)

are needed. As one session starts, the corresponding session identity is generated for the system to track and maintain the overall endpoint behaviours. The advantage of session-type-based specifications is that session types concisely and uniformaly represent all necessary information for concurrent interactions.

The session-type-based approach uses sessions to *partition* the interactions among processes. Although interactions are concurrent, because a session has linked all related interactions, there is no interference between different sessions. Thus, inside a session, the interactions are executed step by step, to which the principles of Floyd and Hoare's works can be applied: Every interaction point attached with assertions (at the sender and the receiver) is viewed as a control point. At the same time, since different sessions concurrently exist in the network, we can focus on exploring and specifying the properties of invariance among sessions. For example, one invariance in our setting is the property of coherence. One rule of coherence states that, in all monitors' specifications, there is one and only one shared channel with input mode, and, whenever there is a sender, there is a receiver.

Other methods of algebraic approaches, like CCS (115), and functional approaches, like the method of Broy (25), attempt to replace state-based reasoning with other proof techniques. In an algebraic approach, verification is based on applying algebraic transformations. In a functional approach, the rules of function applications are used.

Our approach is an algebraic one. We view a specification as a session-type-based language through formally defining its syntax and semantics. We provide the labelled transition system (LTS) of session-type-based specifications. As the operational specifications advocated by Lam and Shankar (97) and Lamport (99), the LTS makes the specifications efficient to specify parts of concurrent processes.

3.2 Specification and Verification for Conversations

In the literature, the purpose of defining specifications for conversations is to coordinate heterogeneous autonomous agents (95, 149). Many works related to this subject are particularly for multi-agent systems (MASs). In general, protocols are specifications for conversations.

There are two purposes of the exploration of conversation specifications: To define policies for conversations and to inspect (statically or dynamically) conversations among

multiple participants. To achieve them, one needs to firstly identify the relationships among the multiple system components, which may possibly locate in different domains. According to Singh's work (149), to figure out distributed coordination of heterogeneous agents, we need to know the skeletons of agents with compact descriptions of the given agents in terms of their events for coordination, and the relationships among the events occurring in these skeletons. They adopted a graphic-approach, Dooley graphs (109), to generate the skeletons and the relationships required for coordination. The skeletons here are *specifications*, while coordinations in multi-agent systems generally are meant to be *interactions* among agents. With the use of Dooley graphs, the table recording the histories of agents in conversation is generated and the skeletons of an agent involving in coordination are produced. This work represented a methodology to produce a specification from observations and given requirements.

Other related works are (95, 150) of Smith et al. (150) provided a comprehensive semantics for the conversational policies through using joint intention theory (JI), which prescribes a way to execute actions jointly by a team of agents. In (95), the most important aspect of a conversation protocol is not the set of communicative actions involved in that protocol but the effects or the states that these actions bring about. They proposed a formal analysis of landmark-based approach to view a conversation protocol as partially ordered landmarks. Each landmark defines a set of specifiable semantic properties that must hold of the agents involved. Based on (150), they represented and analysed families of protocols with JI theory. They expressed JI in terms of joint commitment between two agents to bring about a certain state of affairs which results in appropriate individual commitments.

Wooldridge's work (158) examined the issue of developing semantics for ACL (agent communication languages [1]). It considered verification of conformance to the semantics and had given the first precise definition of what it means for an agent communication framework to be verifiable. Another work of his (159) argued that, for agents in large-scale multi-agent systems to safely communicate to one another, it must be able to determine whether any system that claims to conform to an ACL standard actually does so when agents are possibly built by different organisations using different platforms (136). Particularly, this conformance — the semantics of the language adopting

[1] ACL means either KQML or FIPA ACL, proposed by Foundation for Intelligent Physical Agents (FIPA) in 1997.

by a system conforms to the standard ACL semantics — can be determined by an independent observer. Koning and Oudeyer in (93) proposed POS (Protocol Operational Semantics), a unification and generalisation of interaction protocols, which allows the design of dedicated ACLs together with policies or constraints of messages.

The motivation behind the issues they addressed are similar to ours. They explored the semantic conformance for standard ACL, while we explored for the session-type-based specification languages. ACL is to cater for the needs of agents to interact in a shared language, hiding the details of their internals and to build communities of agents that can tackle problems that no individual agent can. Session types share the same interest. According to the investigation of (96), the existing semantic approaches for verifying the conformance for ACL standard, however, rely on multi-modal logics that are often non-computable and/or have no efficient implementation. Since session types standardise the sequence of actions (including the permutation of actions to cater for asynchronous environments) of a given protocol and the operational semantic system of specifications is deterministic, the semantic approach for verification of communications proposed in this thesis is decidable and computable.

Meng et al. in (113) explored the synthesis of Reo circuit scenario-based (i.e. session-based) interaction specifications, where Reo is a channel-based coordination language. Although their tool (i.e. Reo circuit) is different from ours, both of us aim to systematically generate runtime monitoring mechanism (i.e adaptors in their words) to inspect the behaviour of interactions. They argued about session types are for coping with heterogenous descriptions of component interfaces. By applying session types, an adaptor can be automatically generated from the adaptor specification (specified by session types), which establishes a correspondence between messages in different components. They also argued that the adaptor specification based on session types requires implementation details to achieve the correspondences among methods (and their parameters) of different components. Our responses are: (I) We aim to *specify interactions* and the contents of interaction messages among endpoints, but leave the private tasks, like how to deal with information at different locations, to the local endpoints. (II) Session types are thus useful to cope with runtime verification for the session-level interactions as unexpected runtime endpoint values are generated by endpoints. Our approach focuses on injecting specifications into external monitors, rather than automatically generating endpoint processes which obey specifications. Session-type-based specification gives

the freedom for endpoint programmers to design their own interfaces. Through runtime verification of an external monitor, which is unattached to endpoints, as long as endpoints' behaviours obey what the specifications specified, endpoints and the overall system obey the given global scenario.

3.3 Rumtime Verification for Sharing Memory among Multiple Participants

How to specify specifications for multiple participants (e.g. multiple threads) to share memory in one operation system or in one application, and how to verify the behaviours of multiple participants to see if they obey the specifications, are important topics in the area of specification and verification. In these topics, most works consider the participants and their actions are in *one common domain* (e.g. one operation system), which provides the ability for scheduling processes. Thus, when they consider runtime verification for sharing memory, they are able to design *synchronisation* mechanism to achieve data-race freedom. These works include (98, 138, 142), which aim to explore and propose methods for synchronisation of asynchronous actions (e.g. multi-threads in their cases). They are briefly described in the follows.

(98) proposed the algorithm for synchronising physical clocks, which should always imply logical clocks. If one can resolve the problem of synchronising physical clocks at different domains, then one can apply the similar idea to synchronise the events issued by different local processes, which require, at least, the partial ordering. However, many issues arise in practice as those discussed in (66, 91, 102, 114, 139). Even the physical clocks at different domains can be practically synchronised, preserving the order of messages (or the actions triggered by messages) costs the overall system unefficient because the earlier-arrived but later-issued message needs to wait for the later-arrived but earlier-issued message.

(142) proposed "Eraser" for detecting data races in lock-based synchronisation for multi-threaded programs by checking that all shared-memory accesses follow a consistent locking discipline. Eraser uses binary rewriting techniques to monitor every shared memory reference and verify that consistent locking behaviour is observed. (138) focused on synchronisation for achieving linearisability and refinement to make the use of sequential specifications for data structure operations.

However, synchronisation is not practical in large-scale distributed systems because no scheduler can schedule the events of processes in different independent/dependent domains at the same time. Thus we adopt a different way: Directly cope with the issues of asynchrony and concurrency in large-scale dsitributed systems, rather than designing synchronisation mechamism.

In Chapter 8, we address the issue of sharing memory among multiple heterogeneous parties and realisation of asynchronous monitoring in large-scale distributed systems. Due to the nature of asynchrony, the order of floating messages is not preserved, which implies that the order of actions, which are corresponding to these messages, is not preserved. Although our governance framework focuses on the safety of the session-level by regulating the interactions through runtime monitoring, we also consider the effects when the *states* of endpoints are specified in the specifications of monitors, which may make runtime monitoring ineffective due to the nature of asynchrony. When the states of endpoints are specified, sharing memory in *sessions* is stipulated. The formal syntax and semantics of *stateful* specifications and the theorems for asynchronous specifications are introduced in Chapter 8.

3.4 Dynamic Monitoring Frameworks

Works (73, 107) gave a brief view of dynamic monitoring or runtime monitoring. According to Havelund and Goldberg's work (73), specification-based runtime verification consists of monitoring a program's execution against a user-provided specification of intended program behaviour. In (107), Leucker and Schallhart defined runtime verification as the discipline of dealing with the detection of violations (or satisfactions) of correctness properties. They pointed out the use of runtime verification for contract enforcement.

Schneider in (143) introduced security automata for specifying exactly that class of security policies. It is a runtime monitoring mechanism for enforcing safety properties. (143) characterised that the set of executions for a security policy which is enforceable by security automata is a safety property. This work provided a general picture for runtime monitoring based on specifications or policies. As Deniélou and Yoshida in (51) addressed, every well-formed global session type can be represented by a multiparty session automata (MSA), i.e. it implies that session types are enforceable.

The earliest mechanism of monitors are introduced in the 1972 U.S. Air Force report (11), which was based on the works of Lampson (104). Hoare in (78) invented the name *monitor* based on Brinch-Hansen and Dijkstra's concept (58, 72) and defined it as a collection of local administrative data, together with some procedures and functions which are called by programs wishing to acquire and release resources. It targeted to avoid data races by using lock to realise monitor's mutual exclusion, which means at each time, at most one thread may occupy a monitor.

The monitoring check proposed in (78) is for static-checking: Detecting data races during compile time through reasoning about the semantics of programs. (106) pointed out the limitations when people want to dynamically allocate shared variables. Dynamic monitoring is able to monitor such an environment where (I) a process (e.g. an object, which may allocate new variables) can be created; and (II) there is a monitor for every instance of an object. Each of these monitors shares the same code but has its own lock for process.

In summary, dynamic checking, compared to static checking, enjoys (I) dynamic verification making the whole system becomes scalable with the assurance of correctness even when untrusted components exist, and (II) the floating messages delivered during runtime have no chance to pollute local source code because evil messages have been blocked by a corresponding monitor.

3.4.1 Inlined Monitoring

In inlined monitoring, monitors run during runtime and exactly keep the status of monitored process and make the communication overhead significantly reduced. However, since the monitoring code is written in the code which should be protected by replacing the annotations containing specifications at every control point, an inlined monitor cannot detect the unexpected interruptions.

Since an inlined monitor positions at every local process, the interactions between an endpoint and its corresponding inlined monitor are *synchronous*; therefore, formalising the interactive behaviours among endpoint processes and the corresponding monitors and the external is not a challenge. The works for inlined monitors, such as (6, 71, 151), aimed to provide a policy specification language for policy written in monitors and certify the specifications generated by inlined monitors satisfy the original policy. However, inlined monitors may suffer from the code pollution of malicious endpoints and

become expensive especially when thousands of new and possibly untrusted participants are joining a large-scale distributed system every hour.

(6, 44) introduced a formalisation of monitoring and monitor inlining for the Java Virtual Machine. They specified monitor inlining correctness using annotations, which are based on Floyd's work. Their monitors are security automata, as the one of Schneider's. The monitoring code is embeded into the local code which need protection. They also analysed and specified the synchronisation check point.

Falcone et al. in (61) introduced a generic formalism for runtime enforcement based on the concepts of Hoare's monitor and Schneider's security automata for analysing properties. They studied the problem of enforcement relatively to the so-called Safety-Progress hierarchy of regular properties, which provides a finer-grained classification of enforceable properties. They showed how to generate an enforcement monitor in a systematic way. Although their goal is different from ours, they provided a good summary and explored the composition, synthesis, and enforcement abilities of monitors.

3.4.2 Outline Monitoring

Another spectrum of monitoring framework is *outline* (or offline) monitoring (37, 38, 107), in which monitors are external to application code. Outline monitoring implementation, which may work on a finite set of recorded executions, is decoupled from application code or execution. It represents a desirable feature in untrusted environments, and is commonly used for distributed infrastructures. Outline monitoring better supports decentralised monitor configuration.

Chen and Rosu in (37) introduced monitoring-oriented programming (MoP), a paradigm for combining formal specification with implementation. It provided an outline monitoring framework and light-weighted formal method to check conformance of implementation to specification at runtime. The monitoring code using the same target language for the implementation is automatically generated during a pre-compilation stage. The generated code has the same effect as a logical checking of requirements. In outline monitoring, the monitoring code is executed within a different process, potentially on an independent component (e.g. external monitor). This kind of monitoring allows one monitor to guard multiple endpoint processes. In (38), Chen and Rosu extended their previous work to allow the specifications have parameter, together with

an implementation of JavaMOP that supports parameters. With this extension, it can state and monitor safety properties referring to two or more related objects.

In the above works, since they are not designed to monitor interactive behaviours among endpoints, their formal syntax and semantics of specifications are directly written in past time linear temporal logic, thus logic experts, domain experts, requirement analysers, software designers and programmers are needed to cooperate for generating the monitoring code. On the contrary, session types represent the causal relations, the timing, the steps of taking actions of sending, receiving, and passing messages in a straightforward way, and at the same time, elegantly express the scenario which may be very difficult to be described in temporal logic. Our session-type based specifications can thus be used for generating monitoring code more easily and directly. As an implementation of our formal specifications, Scribble (145), a completely session-type-based description language, is developed to give program developers a clear and easy-to-read endpoint behaviour guidance.

Against to the shortage of inlined monitoring mentioned above, outline monitoring, positioning externally against to endpoint processes, can ideally provide economic and robust monitoring solutions. However, since this setting results in asynchronous interactions among local processes and external monitors, it is rather hard to capture and formalise the interactive behaviour, which is the key for designing effective specifications for monitors. One main contribution of this thesis is that we capture and formalise the asynchronous interactive behaviour for dynamic monitoring based on the outline monitoring setting.

3.4.3 Monitoring for Conversations

The main related works include (10, 148). The runtime monitoring of (148) is based on MSCs (message sequence charts). They adopted MSCs as their specification language to represent the global scenario, and transformed these diagrams to automata to enable conformance checking of finite execution traces against the specifications. Because their monitoring is based on MSCs, every action at endpoints is strictly ordered (i.e. total order), which means, when one endpoint, say p, is waiting for inputs from q_1 and q_2 which are specified as p should firstly input q_1's message then q_2's even though the messages from q_1 and q_2 have no any causal relation, when q_2's message arrives at p earlier than q_1's due to asynchrony, the monitor at p needs to refuse q_2's message. Our

monitoring is more advanced by having *permutation* mechanism to make p's monitor to accept q_2's message if these two messages have no causality. One main contribution of their monitoring is that it ensures liveness for finitely terminating behaviours. But, note that, liveness property generally is not monitorable because interactive behaviours may not terminate.

(10) proposed dynamic monitoring framework based on MPSTs for MASs (multi-agent systems) to guard interactions between local agents and the environments. They gave a procedure for automatically deriving a self-monitoring MAS by the Jason agent oriented programming language to verify that a MAS implementation is compliant with a given *global session type*, which can naturally be represented as cyclic Prolog terms. Another work of theirs (9) proposed global types, which are very similar to global session types for multi-party interactions for dynamic checking without projections.

Compared to ours, the similarity is for guarding interactions during runtime based on session types. But their monitoring is centralised and inlined while ours is completely distributed and outline; and therefore, their monitoring can only realise synchronous monitoring rather than asynchronous one. Most important, (10) focused on implementation of monitors, while we aim to establish a theoretical foundation for dynamic monitoring. The implementation of dynamic monitoring based on the theorems proposed in this thesis can be found in (39), contributed by Pierre-Malo Deniélou. Although (9) investigated the theorems for their framework, they do not have the theorems of local and global safety, local and global transparency, and session fidelity, which are contained and thoroughly proved in this thesis.

3.5 Governance

3.5.1 Service-oriented Architecture (SOA) Governance

Service-oriented Architecture (SOA) (92, 94) is a method of designing and managing systems which are characterised by coarse-grained services and service consumers. SOA governance is however too limited compared to large-scale distributed systems, such as OOI, because its ultimate governance goal is to deliver agility and objectives to businesses (4, 111), and it is the governance mechanism particularly for an enterprise. In other words, it is for regulating processes in one single domain, which has consistent policies, typing rules, and governance goals: This domain may contain many sub-units,

e.g. an enterprise can have many departments, but all these sub-units follow common policies. Our current governance mechanism is general enough for formalising the automatic goveranance parts of SOA governance.

3.5.2 Law-governed Interactions

The series of works (119, 120, 121, 122, 123, 163), for law-governed interaction mechanism, which regards control/policy mechanism in distributed systems, gave a general picture and concept for governing interactions.

However, these works may not be able to capture the properties and theorems hidden behind the complex facts of interactions. Their formal model, called LGI (law-governed interaction), represented the *static relations* between laws and interactions: Defining law-governed interactions as those local interactions obeying global law. Their model did not deepen into the effects of concurrency and asynchrony of interactive behaviours which are the norm in distributed systems. They did not provide formalism based on process algebra to represent the *dynamics of interactions* among the local endpoints, their corresponding monitors, and the network.

Compared to these works, we represent the dynamic relations among endpoints, system monitors, and the network for runtime monitoring, and the formalisation of the dynamics of monitor's specifications. We also explore the issues that, for some intuitive sensible specifications, asynchronous interactions will result them in becoming insensible, and we characterise the specfications which are proper for asynchronous environments.

3.6 Endpoint Access Control

The access of users to shared objects at endpoint is a basic operation in distributed systems. Though certainly many of the works are worth mentioning, one of the early clear statements on access control mechanisms and policies is found in Lampson's (105), which introduced two notions, Protection Domain and Access Control Matrix (ACM). Protection Domain is the environment defining a set of objects (e.g. processes) and the types of operations that can be invoked on each object. An ACM is a matrix saying who can access which with what operations.

We do not provide formalism for detailed access control mechanism for the local, but we formalise a session-level endpoint capability control: We specify the capability of a role involving in a session, which is specified under a particular protocol defining the behaviours of session-roles (the roles in a specific session). Only the endpoint playing a particular session-role can output (resp. input) messages to (resp. from) other particular session-roles. The related works in this catagory include Lampson et al. (103) and Abadi et al. (2), safeDpi (75), typing system for a higher-order π calculus (160), and the series of works of Klaim (46, 47, 48). The frameworks of (103) and (2), are based on logical approach, in which they introduced the *logics* in access control and the famous "say" authentication logic; while others are based on *process* algebraic approach as which this thesis adopts. Others provided formalism for access control by assigning types, i.e. capabilities, to located processes (according to (48, 160)) in distributed systems to express and enforce policies that control access to resources and data. In work (160), they proposed a typing system for a higher-order π-calculus. The typing system is used as an access control for an endpoint to judge the accessibility of processes for protecting local resources.

safeDpi (75), proposed by Hennessy et al., modelled filtering for migrating processes with channel dependent types. With the similar concept of processes migration, Ferrari et al. (63) proposed an ambient-based runtime monitoring formalism, called guardians, which aims for the rights of access control of network processes.

The series works of Klaim (46, 47, 48) advocated a hybrid (dynamic and static) approach to access-control against capabilities including resources and data: The aim is to provide a static checking that is integrated within a dynamic access-control procedure controlling data communication and processes migration. These works (46, 47, 48, 63, 75) also addressed access-control for mobility.

Other related works such as those of Dezani-Ciancaglini et al. (55, 56) provided a type system for the $Xd\pi$ calculus (67), proposed by Gardner and Maffeis, which is a calculus for describing a network of locations, each of which consists of data tree and processes. For a well-typed $Xd\pi$-network, every process properly controls its data (e.g. update or copy) and it can access the data at other locations only when whose security level is equal to or less than its security level.

Our work realises an endpoint capability control in *every session* by passing a session-role, denoted by $s[p]$, viewed as a capability, to every process participanting in session

s to identify (i.e. a session identity plus a role) the endpoint process, and embed assertions (22) into monitors. This embeddedness of assertions enables external monitors to not only verify the interaction messages, but also check if the access requester has the capability (e.g. authority) to access the resource. In our governance framework, interactions in distributed systems are partitioned into sessions, thus is called *session-based* governance. Viewing inter-related interactions as one session unit enjoys governance benefits: (I) It reduces the tasks for endpoint access control. Originally an endpoint inlined monitor needs to check and authenticate thousands of input or output access permissions, with session-based outline monitoring, the session-level (i.e. interaction level) check firstly filters most ill-behaved input or output actions, then leaves those obeying global protocol but critical tasks, e.g. request to access confidential data, to endpoint static checking system. (II) Through passing capabilities, delegation and migration can be realised in our calculus.

In summary, compared to these works, our contributions are the formally founded enforcement of global invariants, such as global safety and session fidelity, and the proposal of decentralised session-level runtime monitoring for interactive behaviours among components written in possibly different programming languages. By incorporating protocols and conversations as two formally founded abstractions for programming and runtime, our framework offers formal assurance of clearly articulated properties for distributed applications.

3.7 The Related Calculi

3.7.1 The Spi-calculus

Abadi and Gordon in 1997 (3) introduced the Spi-calculus, an extension of the π-calculus designed for the description and analysis of cryptographic protocols by embedding encription libraries into the π-calculus. They represented protocols as processes in the Spi-calculus: When all the processes are combined, a protocol is represented. They also stated the security properties in terms of coarse-grained notions of protocol equivalence.

We adopt the π-calculus and session types rather than the Spi-calculus is because we want to focus on governing conversations but leave the authentication tasks to each endpoint in different domains. Since in large-scale distributed systems, every domain may have their own authentication systems which are different from one to another,

applying the Spi-calculus would however hinder the flexibility we want to give to the endpoints, and may make our formalism and analysis become complicated. For example, when two endpoints in different domains interact, different domains may have different encryption libraries or different keys. Using the calculus with encryption and decryption primitives make us hard to describe and analyse the critical interactive behaviours among multiple participants.

Another reason is, to explore fundamental theories for governing conversations, the best way is to start from the original theories with the clearest form (e.g. the π-calculus and session types). This reason also applied for the following related calculus.

3.7.2 Policy for Sessions in the Ambient Calculus

The ambient calculus (33), introduced by Cardelli and Gordon in 1998, is a calculus for describing the mobility of concurrent processes. It introduced a paradigm of mobility where computational ambients are hierarchically structured. It describes where agents are connected to ambients and where ambients move under the control of agents. As a note in (110), the ambient calculus is a useful tool to construct mathematical models for security problems because of its facilities in expressing hierarchies of locations and their mobility.

Garralda et al. in (68) introduced BACI (Boxed Ambients with Communication Interfaces), an ambient calculus with flexible communication policies. BACI makes different communication policies with different parents during computation possible. Moreover, BACI makes communication and mobility become explicit by separating the channels of communication between ambients. Our work, to BACI, is for its session-level's governance. It is a promising future work to integrate our session-based governance for conversations into BACI.

4

Assumptions and Terminologies

As the bases for the later chapters, the assumptions and terminologies of system participants and the communicating instances are introduced, and their relations are explained.

4.1 Principals, Processes, and Endpoints in Systems

Since the term "*system participant*" can be for any entity involving in systems, it is vague for representing entities with particular tasks. Therefore, more specific terms like *principals*, *processes*, *endpoints* are distinctly used here, and each of them is explained as follows:

1. Principals: A system consists of *principals*, which are system components involving in the system. The activities of principals include having interactions with other principals: With system' agents, e.g. accessing data storage through a system agent, or dealing with system-level operations, e.g. scheduling, monitoring, updating policies for systems. In summary, a principal can be a user (e.g. a service contributor, a server, or a client, etc.), a system agent (e.g. a monitor or a database agent, etc.), or an electronic device, etc.

   Since interactions among principals are the norm in large-scale distributed systems, principals should be abstracted to realise *communications*. A principal may have many names for different activities, but is identified with a unique identity

such that (I) it has its own data storage, which may be positioned at the endpoint or at a shared database in the network; (II) it has *pre-defined* capability of executing particular activities and communications; (III) it is viewed as *local* in distributed systems. All of them can be referred to a unique principal with its identity.

2. Processes: With standard definition of processes, a process is an instance of a program of an application that is being executed. A process may interact with other processes and have multiple threads for executing multiple tasks. A principal can have more than one process running applications for it, while a process can concurrently include more than one *endpoint* by having multiple threads. Since we focus on the conversations (sessions) among applications, we generally assume a process involving in sessions have interactions with other processes.

3. Endpoints: An *endpoint process* is for a local process, and an *endpoint* is for a *session participant*, which is represented by a shared channel (name) linking connections to it. Although many works use "endpoints" for "endpoint applications", in distributed systems, an application may concurrently run more than one system component. In this thesis, an endpoint is abstractly represented by a shared name with the assumption that a unique input-mode shared channel connects communications to it. A process can have more than one endpoint by having distinct shared names. Thus, a process can execute more than one activity simultaneously through different session endpoints.

Figure 4.1 illustrates the relationships between principals, processes, and endpoints, which are connected by shared channels a, b, c, d, e and f.

(i) There are Principals A and B. A principal can have more than one process. In the figure, Principal A has an Application A_a, which has two processes Process A_1 and A_2 running concurrently. Principal B has two applications, which are Application B_a and B_b. Each of them has one process. Application B_a has Process B_1, and Application B_b has Process B_2.

(ii) A process can have more than one endpoint. In the figure, Process A_1 uses three endpoints, connected by shared names a, b and c. These shared names represent

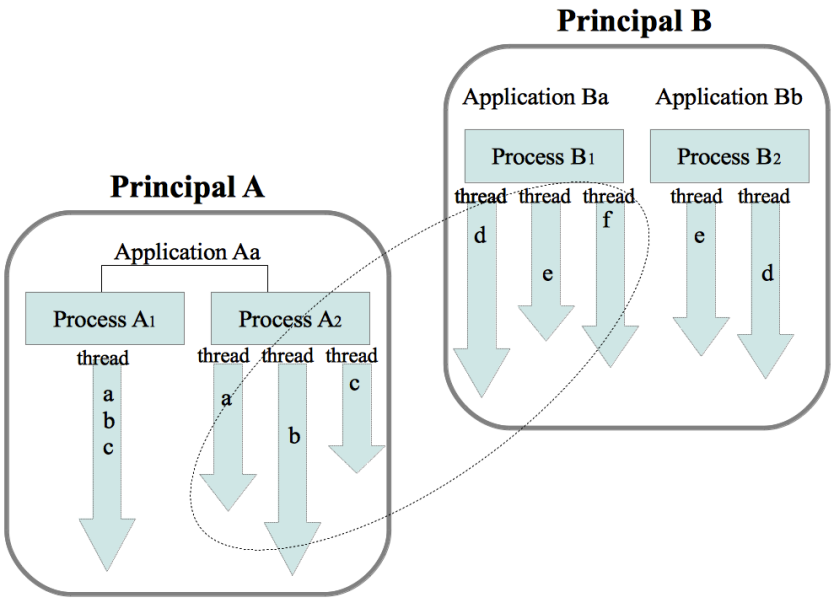

Figure 4.1: The Relationships between Principals, Processes, and Endpoints

the endpoints. Process A_2 also uses these three endpoints but running separately in three concurrent threads. Process B_1 uses three endpoints d, e and f separately for running in three concurrent threads. Similarly, Process B_2 uses endpoints e and d separately for running two concurrent threads. Note that Process A_2 and B_1 are in one session where interactions are linked by endpoints a, b, c of Process A_2, and d, e, f of Process B_1.

(iii) An endpoint is represented by a shared name which realises a connection for this endpoint and represents the usage of this endpoint. For example, Principal A has endpoints represented by names a, b, and c for different activities in two different processes. a in Process A_1 is used for running an activity, while a in Process A_2 is used for participating in a session (to interact with Process B_1). Similarly, Principal B has endpoints represented by names d, e and f.

In summary, a system principal may run several network applications (executed by processes) concurrently, where each application can possibly have more than one proxy endpoint for communications. Those applications belong to the same principal identity.

4.2 Communications and Interactions

1. Communications: A "communication" is a piece of *interactive behaviours* in a session, which can be realised by a system component's one-way activity to another system component, e.g. sending a message with particular *content(s)* to another system component, or receiving a message from another system component.

2. Interactions: An "interaction" is a pair of communications. It at least consists of two activities which are dual to each other among two system components communicating to each other. In other words, while a system component (as a sender) sends a message to another system component (as a receiver), the recipient always receives it (synchronously) or is going to receive it (asynchronously). Note that system components are local processes with multiple (session) endpoints.

4.3 Names, Sessions, Shared Channels, and Session-roles

1. Names: We use a name for a communication channel. In this thesis, we classify channels to two kinds: A session name and a shared name. Generally a session name is for identifying a session, but widely speaking, a session-role is also a kind of name for representing a session endpoint's communication capability. A shared name is for a shared point. An input-mode shared name (introduced later) links connections to an endpoints.

2. Sessions: A session, also called a conversation, is a sequence of interactions. In our system, it is identified by a session name, and specified under a particular protocol, which can be represented by a session-type-based specification. A session links related interactions from pieces to a whole. One virtue of using *session* as a unit for formalising thousands of interactive behaviours among endpoints is that a session *structures* interactive behaviours and, based on this structure, provides a basis for analysing and verifying runtime behaviours. When a session name is created, a session with a new session identity is created in systems.

3. Shared names: A shared name denotes a *shared channel*, which is identified by a particular name. It abstractly represents a communication endpoint. It has three kinds: `I` is for input-mode, `O` is for output-mode, and `IO` is for both input- and output-modes. With the explanation of the definition of endpoints, a shared name with input-mode, a mode containing mode `I`, connects input communications to a unique session endpoint, thus, in this thesis, such a shared channel abstracts an endpoint. As a process has an endpoint connected by input-mode shared channel, this process *listens* to incoming messages through this channel but does not send out messages through it; as a process is connected by a shared name, say a, with output-mode `O`, we say this process uses channel a to *send out* messages to the process having common shared channel a but with mode `I` or `IO`. The relationship between `I`-, `O`-, and `IO`-mode shared names is illustrated in Figure 4.2. Note that, a shared name with mode `I` or `IO` represents an endpoint of a process; it *locates* this process and makes others be able to send messages (i.e. output messages) to it (i.e. input message from others). On the other hand, when a process has a shared name with mode of `O` or `IO`, the shared name represents a *knowledge* about where

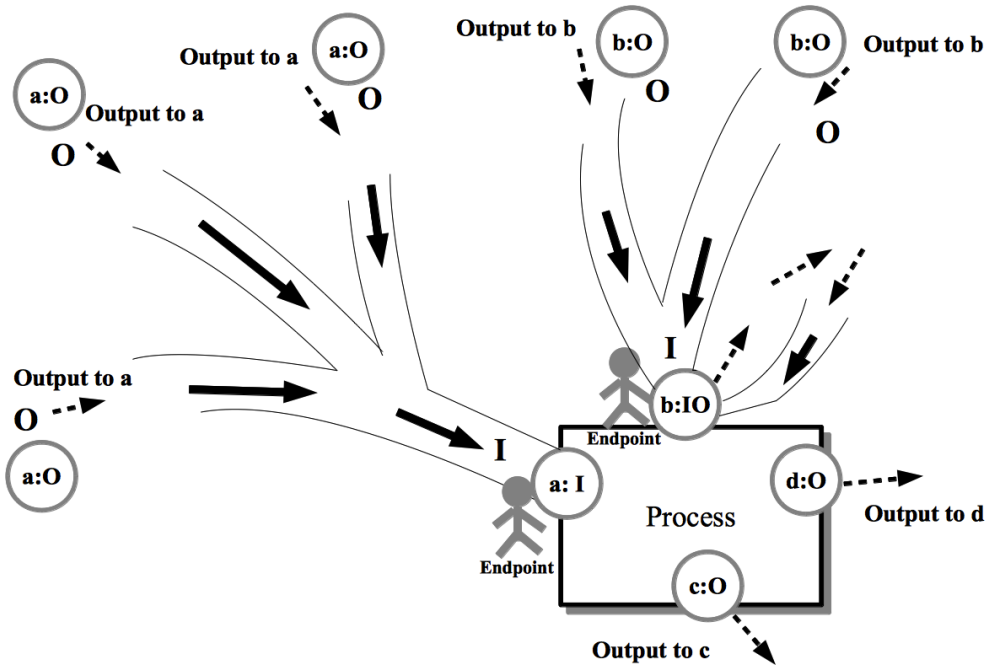

Figure 4.2: The Relationship between Input- and Output-modes Shared Names

(i.e. to whom) it can output a message to the target receiver; this process itself is driven by other endpoints represented by other input-mode shared names. Note that, for flexibility, a shared name representing an endpoint can have both modes `I` and `O`, called `IO`-mode, denoting that this endpoint is able to communicate internally through itself. For example, the endpoint b : `IO` in Figure 4.2 can output a message to itself. Also note that, an input-mode (i.e. `I`- or `IO`-mode) shared name locates an endpoint at a particular locaton (e.g. at a principal) in the network.

4. Session-roles: A session-role is played by an endpoint with the shared name that abstractly represents this endpoint. A session generally includes more than one endpoint, and a local process can participant in more than one session by having

multiple endpoints in it. For exampe:

$$P = s_1[buyer, seller]!\langle 10\rangle \mid s_2[trader, bank]?(x).P'$$

process P plays role *buyer* in session s_1 by an endpoint, say a, and concurrently, plays role *bank* in session s_2 by another endpoint, say b. Since a session is specified by a particular protocol, which defines roles playing in this protocol and confines the behaviours of roles for interactions, every role in a session is *pre-defined*. A role in a session is generally called a *session-role*. As an endpoint driven by a process *involves* (i.e. joins) in a session, it plays one particular session-role connected by a shared channel.

4.4 More Illustrations for Shared Names

By the assumption, an `I`- or `IO`-mode shared name locates an endpoint and abstractly represents the endpoint. More illusrations below concretely explain how an input-mode shared name connects communications to an endpoint.

Figure 4.3 shows that a process can use an input-mode shared name to participate different activities, but can not use an `O`-mode shared name for this purpose. An `O`-mode shared name in a process is a knowledge telling the process where to output a message. In Figure 4.3:

- Process A uses `IO`-mode shared name a to play role $Bank$ specified under the protocol G_1 and has the knowledge of where to output to $Bank$ specified under protocol G_2 by having $c : \mathtt{O}(G_2[Bank])$, and the knowledge about the $Buyer$ specified under protocol G_3 by having $b : \mathtt{O}(G_3[Buyer])$. For convenience, we say Process A's endpoint is represented by a, or simply say its endpoint is a. Note that, although shared names b and c also appear in Process A, they are output-mode shared name and can not represent an endpoint process.

- Process B uses `I`-mode shared name b to play role $Buyer$ specified under protocol G_3. As explained above, the `O`-mode shared name d for role $Bank$ specified under protocol G_1 can not represent an endpoint in Process B. Since Process A has `O`-mode shared name b with $b : \mathtt{O}(G_3[Buyer])$, Process B and A can communicate to each other through channel b. Although Process A has `IO`-mode shared name a of

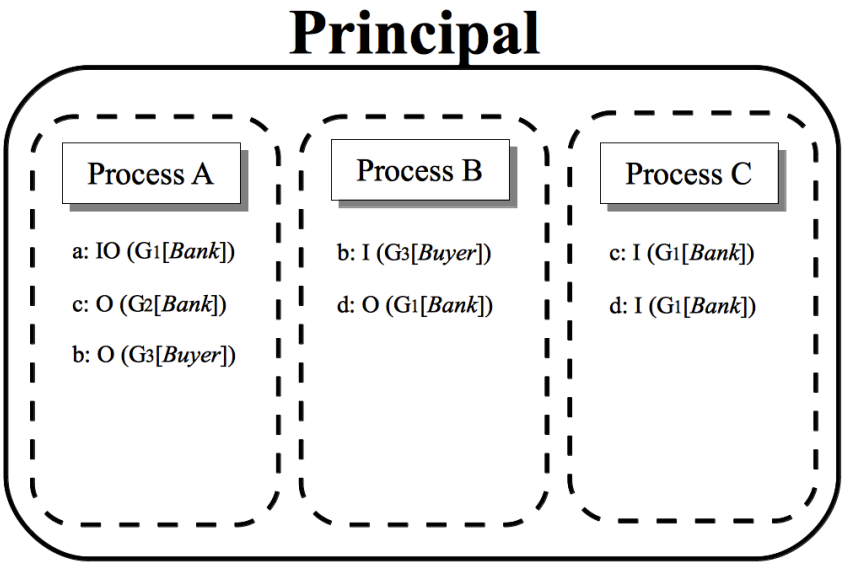

Figure 4.3: Endpoints Represented by `I`-mode Shared Names in a Principal

playing role *Bank* specified under protocol G_1 and Process B has `O`-mode shared name d of role *Bank* specified under the same protocol, they cannot interact with each other because they do not have a *common* shared name; in other words, they cannot be in one session.

- Process C uses `I`-mode shared name c to play role *Bank* specified under protocol G_1. Process C also uses `I`-mode shared name d to play role *Bank* specified under the same protocol. Thus Process C has two endpoints represented by c and d respectively. Note that the same role specified under a common protocol can be played by different `I`-mode shared names. Also note that, Processes C and B have interactions through shared channel d, while Processes C and A cannot have interactions through c because they are bound to different protocols.

The follows give the example of showing the relationships between an input-mode and an output-mode shared name. Assume an email server providing mailing service

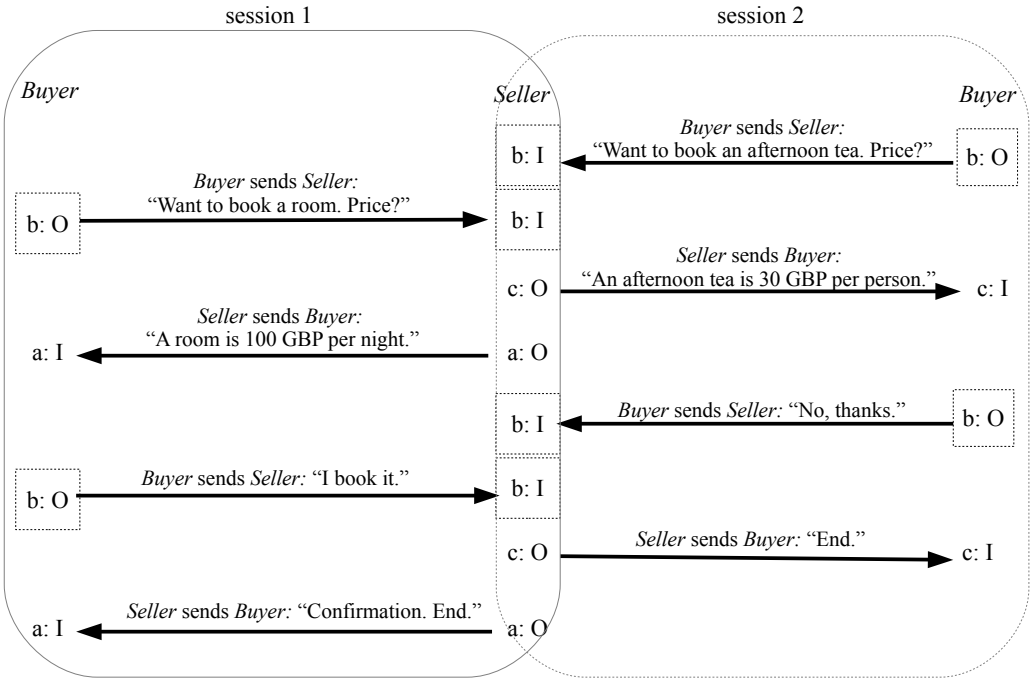

Figure 4.4: Using Shared Name to Represent Endpoints

can be abstracted by $a : \mathtt{I}(G[Server])$, which denotes the server, playing role $Server$, specified under protocol G, is using shared name a as an endpoint to *input* communications. When a client has a *knowledge* denoted by $a : \mathtt{O}(G[Server])$, this client can send a request (output) to this server via channel a. Assume clients $C_1, ..., C_n$ have `I`-mode shared channels $c_1, ..., c_n$, e.g. $c_1 : \mathtt{I}(G[Client]), ..., c_n : \mathtt{I}(G[Client])$, playing role $Client$ with mode `I`, specified under the common protocol G, to receive the server's responses. A server can response to them because it knows "where the clients are" by *having* knowledge of $c_1 : \mathtt{O}(G[Client]), ..., c_n : \mathtt{O}(G[Client])$, with mode `O` under the matched protocol G.

Figure 4.4 illustrates another example where endpoint seller uses `I`-mode shared name b to play role $Seller$ in session 1 and session 2. For role $Seller$, channel b is *uni-direction*, which is used for inputting messages from the outside. The role $Buyer$ in session 1 and the role $Buyer$ in session 2 are run by different endpoints represented by different shared names: One is run by `I`-mode shared name a, another is run by

`I`-mode shared name c. Both of them have `O`-mode shared name b, which means they both know where the *Seller* is. Similarly, in order to let the *Seller* know where they are, they provide their `O`-mode shared names a and c when they established sessions.

In the network, an `O`-mode shared name can appear in more than one principal, but an `I`- or `IO`-mode shared name can only appear in one principal. If we allow an `I`-mode shared name to appear in more than one principal, it makes the routers hard to route service requests to different principals who have common input-mode shared names but are located in different domains. Also, the security issue arises because the principals with common input-mode shared names are not specified and makes it hard to realise who should take responsibility when something goes wrong with a cooperative common task. If we want to describe that a common task can be done by more than one agent, this scenario can be realised by delegating the task to agents with different shared names (see Example 5.3.1). For simplicity, this thesis assumes that a shared name *having* mode `I` or `IO` can only appear in one principal in the network. More discussions about the design of `I`-, `O`-, or `IO`-mode shared name to cater for monitoring and routing purposes can be found in Appendix A.2.

5

The Capability-passing π-calculus

Overview In this chapter, we introduce the formal syntax and semantics of the capability-passing π-calculus. It is a process calculus for *dynamic governance* for large-scale distributed systems, where basic interaction steps are asynchronous message exchanges, and interactions are concurrent. With the primitive nature of operations, the calculus leads to a clear monitoring mechanism, introduced in Chapter 7.

We start from Section 5.1 with a motivating example. In Section 5.2, we introduce a real-life use case from OOI, an under-construction large-scale distributed system, to show the expressiveness of the proposed calculus. The calculus is formally introduced in Sections 5.3 and 5.4. Section 5.3 provides the syntax and semantics for local processes; while Section 5.4 provides the syntax and semantics for the network. Local processes and the network together compose large-scale distributed systems that we target.

5.1 Motivating Example

The capability-passing π-calculus has fine-grained primitives for system entities to *activate or deliver capabilities* via a session in which they involve.

This calculus has two main concepts. One is viewing a *type of session* as a template so that participants can reuse it, e.g. participants can do tradings by establishing a *Buyer-Seller-Broker type* session where roles $Buyer$, $Seller$ and $Broker$ are defined and the interactions among those roles are described in such a session. A Buyer-Seller-Broker type session can be realised as a session that follows Buyer-Seller-Broker protocol. A

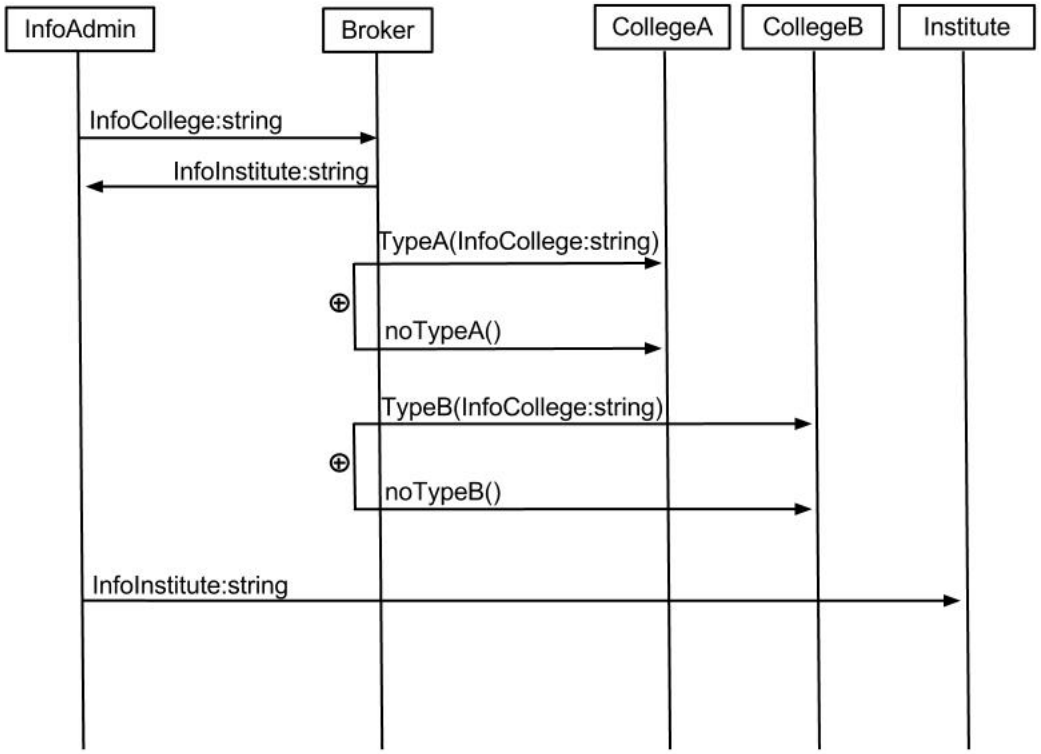

Figure 5.1: Delivering Information Through a Broker to Proper Targets.

session is a collection of *structured interactions* and the *roles*, generally called *session-roles*, involving in those interactions. Another concept is viewing a role defined in a session type as a *capability*. When a participant can play a session-role, he or she has the *capability* to execute the series of behaviours defined in this session.

These concepts come from the observation of activities of system entities in large-scale distributed systems, where the activities may be only available to some particular participants, e.g., when a user only registers himself/herself as a role $Buyer$, in some activities, he/she can only behave as a $Buyer$ but not as a role $Seller$. As for an entity identified by a particular system ID, it should always be able to flexibly communicate with others through joining different session-roles concurrently, or delegating/delivering its capabilities to others for playing these session-roles.

A session-role arises as a basic syntactic entity in this calculus. The mechanism for playing different roles in different sessions concurrently makes system entities be able to flexibly and simply represent various forms of multi-party session establishment. Before

formally introducing the syntax and semantics of our calculus, we use Figure 5.1, a message sequence chart, to show that the benefits as a session can be established through *asynchronously* linking each session-role. Assume there are five system participants, who join a conversation, which involves activities among session-roles $InfoAdmin$, $Broker$, $CollegeA$, $CollegeB$ and $Institute$. As Figure 5.1 illustrates, $InfoAdmin$ is for information announcement, $Broker$ is for delivering information from $InfoAdmin$ to proper colleges, which are classified into two levels: $CollegeA$ is for a level-A college, and $CollegeB$ is for a level-B college. $Broker$ will send InfoInstitute to $InfoAdmin$ once it receives the message from $InfoAdmin$. Then $InfoAdmin$ will send a message to $Institute$. The symbol $\oplus$ in the figure means alternative branches which are the choices at $Broker$: if the information is for $CollegeA$, $Broker$ sends to $CollegeA$ by selecting branch TypeA, and sends to $CollegeB$ by selecting branch noTypeB with no content.

Assume, participants playing roles $InfoAdmin$, $Broker$ and $Institute$ are ready for interactions, but not participants of $CollegeA$ and $CollegeB$, which means roles $CollegeA$ and $CollegeB$ have not yet been played by any participant. If a conversation can only be established through *synchronously* binding each session-role (34, 86), then $InfoAdmin$, $Broker$ and $Institute$ cannot start their interactions (even if they are ready) because they are linked in a session and can only start when this session exists; however, this session does not exist if session-roles $CollegeA$ and $CollegeB$ are not ready. It makes participants playing $InfoAdmin$, $Broker$ and $Institute$ become idle and inefficient. On the contrary, if a session can be established through *asynchronously* linking session-roles one by one, then the interactions among $InfoAdmin$, $Broker$ and $Institute$ can set up to start interactions without waiting for participants of $CollegeA$ and $CollegeB$ getting online. The next section will use a real-life example represented by the formal syntax of the capability-passing π-calculus.

5.2 Use Case: OOI Instrument Commands

Examining real-life examples is useful for clarifying the requirements and providing a shape for the formalism. This section uses OOI instrument commands as a use case and applies the formal syntax of the capability-passing-π calculus (introduced in Section 5.3.1) to represent it.

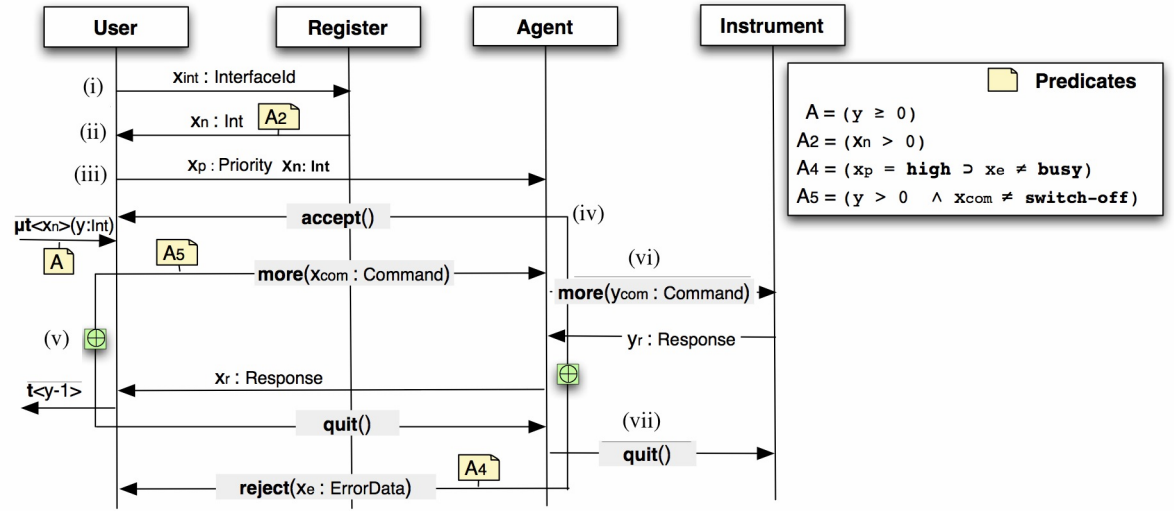

Figure 5.2: Illustration of the Global Protocol for OOI Instrument Commands

5.2.1 The Overview of OOI Instrument Commands

Consider a scientist, who registers in OOI, may wish to use a remote instrument, say a seabed camera which is realised following a specific *application level protocol* (i.e. global protocol). To simplify and generalise this scenario, assume the interactions are realised following the protocol, *instrument commands*, which is illustrated by Figure 5.2 as a message sequence chart. This protocol is specified as a global specification G_{IC}, which allows a user to perform operations on a remote instrument. G_{IC} models a session which involves roles $User$, $Register$, $Agent$ and $Instrument$. The formal syntax of global specification G will be introduced in Chapter 6. Note that, as explained in Chapter 4, a session participant is a session endpoint, also called a session-role.

5.2.2 Specifying Protocols for Instrument Commands

In Figure 5.2, $User$ performs one or more commands on a remote $Instrument$. $Register$ is used to retrieve information on the instrument's usage, e.g., to determine the maximum number of commands allowed in the current session depending on the system load. $Agent$ interfaces the communications with the actual $Instrument$. Full arrows represent interactions where one party sends a branch label and a message content (or just one of the two) to another party. The labels carry information on the branch to follow. Arrows linked by $\oplus$ represent alternative branches.

The conversation proceeds as follows: (i) $User$ sends $Register$ a message $\mathsf{x_{int}}$ of type InterfaceId. (ii) $Register$ replies with an integer $\mathsf{x_n}$ which determines the number of commands that $User$ will be allowed to perform on the instrument. The predicate annotating this interaction specifies an obligation for $Register$ to send a value for $\mathsf{x_n}$ satisfying $\mathsf{x_n} > 0$; dually $User$ can rely on this fact. (iii) $User$ sends $Agent$ a priority $\mathsf{x_p}$ (e.g., *low*, *high*), and $\mathsf{x_n}$ that he or she received from $Register$. We assume that there is a public key inserted in $\mathsf{x_n}$ and $Agent$ can verify if it is a correct value sent from $Register$. (iv) $Agent$ sends $User$ a label which is either `accept` or `reject`. In case of `reject`, $Agent$ sends also an error message $\mathsf{x_e}$ of type errData and the protocol terminates. The predicate for this branch ensures that a request will not be rejected due to the fact that the instrument is busy if the priority is high. In case of `accept` the protocol continues with a recursion $\mu \mathsf{t}\langle \mathsf{x_n} \rangle (\mathsf{y} : \mathsf{Int})$ where y is a parameter initialised to $\mathsf{x_n}$, $\mathsf{y} \geq 0$ is an invariant and y is used to enforce $User$ to perform at most $\mathsf{x_n}$ commands on the instrument. (v) $User$ selects either branch more and sends a new command to $Agent$, or quit and terminates. The predicate $\mathsf{y} > 0 \wedge \mathsf{x_{com}} \neq \mathsf{switch\text{-}off}$ is a guard to the branch more: a new command can be sent only if $User$ has not performed $\mathsf{x_n}$ and anyway the command must not ask to switch off the instrument. Next, either the command (vi) or the quit notification (vii) are forwarded by $Agent$ to $Instrument$. In the former case, $Instrument$ responds to $Agent$ (who forwards the message to $User$).

5.2.3 Asynchronous Session Creation for Instrument Commands

Based on the scenario described in Figure 5.2, an example of session creation of processes is illustrated with formal syntax. The form $s[p]$, called session-role, means role p in session s. For example, $s[User], s[Agent], s[Register]$ are the session-roles representing roles $User$, $Agent$ and $Register$ in session s respectively.

Consider the following process P_1 of $Principal_1$, which creates a session s with specification G_{IC} and sends invitations to others:

$$\begin{array}{rcl} P_1 & = & P_{\text{NEWS}} \mid P_{\text{INV}} \\ P_{\text{NEWS}} & = & \overline{a_2}(s[Register] : G_{IC}).\mathsf{join}(s[User]).P_{user} \\ P_{\text{INV}} & = & \overline{a_3}\langle s[Agent] : G_{IC} \rangle; \overline{a_4}\langle s[Instrument] : G_{IC} \rangle \end{array}$$

where $\overline{a_2}(s[Register] : G_{IC}).\mathsf{join}(s[User])$ represents that P_1 creates session s which obeys to global protocol G_{IC}, and will later join the role $User$ defined in G_{IC}; while

$s[Register] : G_{IC}$ specifies that $Register$ is defined in protocol G_{IC} and obeys this protocol. Similarly for $s[Agent] : G_{IC}$ and $s[Instrument] : G_{IC}$. Their projections, e.g. $G_{IC} \upharpoonright Register$, $G_{IC} \upharpoonright Agent$, and $G_{IC} \upharpoonright Instrument$ are viewed as *capabilities* describing what activities and behaviours can/should have for these roles. The rules of endpoint projection are defined in Definition 6.5.1, Chapter 6.

Since P_1 is the initiator of session s, i.e. currently session s is known and only known by P_1, session s in P_{NEWS} and P_{INV} therefore appear. P_1 sends a *bound* request (i.e. bound invitation), $\overline{a_2}(s[Register] : G_{IC})$, to endpoint (connected by) a_2 for playing session-role $Register$ as the first session invitation, and joins the session as $User$ by $\mathsf{join}(s[User])$; she concurrently sends invitations to others through doing

$$\overline{a_3}\langle s[Agent] : G_{IC}\rangle; \overline{a_4}\langle s[Instrument] : G_{IC}\rangle.$$

Note that, only the *first session invitation* is *bound*, denoted with the shape $\overline{a}(s[p] : G)$ i.e. round bracket, because a session name, s, is newed simultaneously; the later invitations for the same session are *free*, denoted with the shape $\overline{a}\langle s[p] : G\rangle$ i.e. angle bracket, because the session name s has been introduced in $\overline{a_2}(s[Register] : G_{IC})$. Also note that, as $\overline{a_2}(s[Register] : G_{IC})$ has been sent out (from P_1) to the network,

$$\overline{a_2}(s[Register] : G_{IC}).\mathsf{join}(s[User]).P_{user} \mid P_{\text{INV}} \xrightarrow{\overline{a_2}(s[Register]:G_{IC})} (\nu s)(\mathsf{join}(s[User]).P_{user} \mid P_{\text{INV}}),$$

which means, as it informs the network that new session name s is created by doing $\xrightarrow{\overline{a_2}(s[Register]:G_{IC})}$, s is no more only known by P_1 but may be known globally; therefore, for this runtime process, now (νs) is needed to hide session s for private usage (i.e. s is only shared by those P_1 wants to invite). $Principal_1$ is inviting other four principals, $Principal_2, Principal_3$ and $Principal_4$. Shared channels, e.g. $a_2, a_3, \ldots$, determine which invitations each process is entitled to receive, e.g. only $Principal_4$ and $Principal_5$ below can accept (by any other process) an invitation $s[Instrument]$, through a_4 and a_5, respectively. Note that, $Principal_4$, which receives the capability of $s[Instrument]$,

passes this capablity to $Principal_5$ who joins this session to play role $Instrument$:

$$\begin{aligned}
P_2 = &\ a_2(y_2[Register] : G_{IC}).\mathsf{join}(y_2[Register]).P_{register} \\
P_3 = &\ a_3(y_3[Agent] : G_{IC}).\mathsf{join}(y_3[Agent]).P_{agent} \\
P_4 = &\ a_4(y_4[Instrument] : G_{IC}).\overline{a_5}\langle y_4[Instrument] : G_{IC}\rangle \\
P_5 = &\ a_5(y_5[Instrument] : G_{IC}).\mathsf{join}(y_5[Instrument]).P_{instrument}
\end{aligned}$$

With the design of the grammar, system can observe the exchange of capabilities between their endpoint and the network.

5.2.4 The Complete Formal Processes of Instrument Commands

We assume $Principal_1$ uses some local functions $needmore()$ (which determines if the user needs to perform more commands) and $next()$ (which returns a command). Noticeably $\mathbf{0}$, generally omitted, denotes termination. $\mu X\langle x_n\rangle(y).P'$ is the recursion where X is the recursion body defined in P', and x_n is the initiation value and y is the parameter acting as a binder over P'. In P^u_{acc}, when x_n is replaced by 10 received from $Register$, y is replaced by the value of x_n. If $needmore() \cap y > 0$ is true, the process executes to the recursion call $X\langle y-1\rangle$, then X leads the process go back to the beginning of the recursion, where parameter x_n should be replaced by v if $y - 1 \downarrow v$. Otherwise, the process executes $s[User, Agent]!\texttt{quit}\langle\rangle$ then terminates.

$$\mathsf{new}\ a_1 : \texttt{I}(G_{IC}[User]).P_1$$

$$\begin{aligned}
P_1 = &\ (\overline{a_2}(s[Register] : G_{IC}).\mathsf{join}(s[User]).P_{user}) \mid P_{\text{INV}} \\
P_{INV} = &\ \overline{a_2}\langle s[Register] : G_{IC}\rangle; \overline{a_3}\langle s[Agent] : G_{IC}\rangle; \\
&\ \overline{a_4}\langle s[Instrument] : G_{IC}\rangle; \mathbf{0} \\
P_{user} = &\ s[User, Register]!\langle v_{int}\rangle; s[Register, User]?(x_n). \\
&\ s[User, Agent]!\langle \texttt{High}, x_n\rangle; s[Agent, User]?\{\texttt{accept}().P^u_{acc},\ \texttt{reject}(x_e).\mathbf{0}\} \\
P^u_{acc} = &\ \mu X\langle x_n\rangle(y).\mathsf{if}\ needmore() \wedge y > 0\ \mathsf{then}\ s[User, Agent]!\texttt{more}\langle next()\rangle; \\
&\ s[Agent, User]?(x_r).X\langle y-1\rangle\ \mathsf{else}\ s[User, Agent]!\texttt{quit}\langle\rangle; \mathbf{0}
\end{aligned}$$

It follows the process of $Principal_2$ who defines channel a_2, receives an invitation and joins the session as $Register$. Assume $Principal_2$ returns always 10 allowing all participants to perform at most 10 commands on the instrument.

$$\mathsf{new}\ a_2 : \texttt{I}(G_{IC}[Register]).P_2$$

$$\begin{aligned} P_2 &= a_2(y_2[Register] : G_{IC}).\mathsf{join}(y_2[Register]).P_{register} \\ P_{register} &= y_2[User, Register]?(x_{int}).y_2[Register, User]!\langle 10\rangle; \mathbf{0} \end{aligned}$$

Below, $Principal_3$ defines channel a_3, receives an invitation and joins the session as $Agent$. Assume $Principal_3$ relies on a local function $error()$ returning an error data. The agent simply forwards the command and the response between the user and the instrument.

$$\mathsf{new}\ a_3 : \mathtt{I}(G_{IC}[Agent]).P_3$$

$$\begin{aligned} P_3 &= a_3(y_3[Agent] : G_{IC}).\mathsf{join}(y_3[Agent]).P_{agent} \\ P_{agent} &= y_3[User, Agent]?(x_p, x_n).\mathsf{if}\ x_p = \mathtt{High}\ \mathsf{then}\ y_3[Agent, User]!\mathtt{accept}\langle\rangle; P^a_{acc} \\ &\quad \mathsf{else}\ y_3[Agent, User]!\mathtt{reject}\langle error()\rangle; \mathbf{0} \\ P^a_{acc} &= \mu X\langle x_n\rangle(y).y_3[User, Agent]?\{\mathtt{more}(x_{com}).P^a_{com}, \\ &\quad \mathtt{quit}().y_3[Agent, Instrument]!\mathtt{quit}\langle\rangle; \mathbf{0}\} \\ P^a_{com} &= y_3[Agent, Instrument]!\mathtt{more}\langle x_{com}\rangle; y_3[Instrument, Agent]?(y_r). \\ &\quad y_3[Agent, User]!\langle y_r\rangle; X\langle y - 1\rangle \end{aligned}$$

Below, $Principal_4$ and $Principal_5$ define shared channels through which they receive an invitation for role $Instrument$. $Principal_4$ forwards the invitation to $Principal_5$ and terminates. $Principal_5$ joins the session and recursively listening to the message sent from $agent$. The recursion $\mu X.P''$ means there is no parameter or binder over the recursion body X defined in P'', so noticeably there is no parameter of X. If branch `more` is selected, the process (playing $instrument$) returns $response(y_{com})$ to $Agent$ by relying on a local function $response$, and then go back to the beginning of the recursion body. Otherwise, it terminates.

$$\mathsf{new}\ a_4 : \mathtt{I}(G_{IC}[Instrument]).P_4 \qquad \mathsf{new}\ a_5 : \mathtt{I}(G_{IC}[Instrument]).P_5$$

$$\begin{aligned} P_4 &= a_4(y_4[Instrument] : G_{IC}).\overline{a_5}\langle y_4[Instrument] : G_{IC}\rangle; \mathbf{0} \\ P_5 &= a_5(y_5[Instrument] : G_{IC}).\mathsf{join}(y_5[Instrument]).P_{inst} \\ P_{inst} &= \mu X.y_5[Agent, Instrument]?\{ \\ &\quad \mathtt{more}(y_{com}).y_5[Instrument, Agent]!\langle response(y_{com})\rangle; X, \\ &\quad \mathtt{quit}().\mathbf{0}\} \end{aligned}$$

5.3 The Calculus of Local Processes

This section introduces the syntax and semantics of the capability-passing π-calculus for local processes with distributed session primitives. As shown in Section 5.2, the fine-grained scheme calculus is expressive to represent real-world examples, including

those in (35). As synchronous process calculi can not fully identify the properties in distributed systems because the assumptions of synchronous interactions are:

1. A message can be outputted only when its input side (a receiver is ready to receive this outputted message) exists; and

2. An output can be *immediately* absorbed by the input side, which makes the order of messages floating around networks preserved,

this thesis focuses on the processes in *asynchronous environment*, which reflects the reality of distributed systems. Note that, considering the interactions between processes are synchronous or asynchronous becomes critical as we consider stipulating *states*, i.e. the current value of a specific field of an endpoint, in specifications. This issue will be explored in Chapter 8.

5.3.1 The Syntax of Local Processes

The *capability-passing π-calculus* is defined in Figure 5.3. Let $(P, Q, \ldots)$ denote process variables. Session names are identified by $(k, k', \ldots)$, which represent both session instances and session variables. Shared names are identified by $(u, u', \ldots)$, which represent both shared channels and shared channel variables. Let $(p, q, \ldots)$ range over *roles* involving in a session, and $(x, y, \ldots)$ range over *variables*; $(v, v', \ldots)$ range over *values*, including shared name with I/O type, a boolean, or a constant `c`. A constant can be a number, a name, an alphabet, or a string; $(e, e', \ldots)$ are for expressions, while $(S, S', \ldots)$ are for sorts, representing data types of expressions. $(U, U', \ldots)$ is a kind of sorts, particularly for representing how a local process is able to *use* a shared name. *Interaction variables* are those variables that are exchanged in interactions. There are three modes for shared name: `I` for input only, `O` for output only, and `IO` for both input and output.

Rule [$P-$n creation] says a process, with primitive `new` $a : \mathtt{im}(G[p])$, creates a fresh input-mode shared name a in P. Through a, an endpoint can play role p specified under global protocol G. When there are runtime monitors, the fresh name with input-mode should be ensured to be new to the systems. A shared name with mode `O` can not be created by a process but be propagated from one endpoint process having the common shared name with input-mode to another process. For example, assume there exists

a process having shared name a with type $\mathtt{IO}(G[p])$. This process can propagate this name by sending it to other processes; *but* as others receive it, it should be typed by $\mathtt{O}(G[p])$ at these processes. Again, this can be ensured when there are runtime monitors. When a process creates a new shared name with mode $\mathtt{I}$ or $\mathtt{IO}$, it is treated as a bound (hidden) name with the scope including herself and other participants having a with mode $\mathtt{O}$.

The primitives $\overline{u}(s[p] : G)$ and $\overline{u}\langle s[p] : G\rangle$, $u(y[p] : G)$ and $u\langle y[p] : G\rangle$, embeding specification G, make the *control of session creation* possible. The actions of *bound* request and accept with round bracket introduces a *new session* to the local process. For example, when $P_1 = \overline{a}(s[p] : G).P_1'$, P_1 creates a fresh name s and simultaneously invites endpoint a for playing role p. $\overline{a}(s[p] : G)$ introduces new session s to P_1' and s acts as a binder over P_1'. Similarly, when $P_2 = a(s[p] : G).P_2'$, session s, which is *new* to P_2', is *created* and simultaneously P_2 accepts the invitation for playing role p in session s. The actions of *free* request and accept with angle bracket mean that s is not fresh to the processes. As the example shown in Section 5.2, only the *first* request and accept are *bound*.

$s[p]$ is viewed as a *capability* of having behaviours of playing role p specified under global specification G. Either $u(y[p] : G).P$ or $u\langle y[p] : G\rangle; P$ shows an endpoint has the capability of playing role p specified under G and then behaves as P. A special message *invitation*, denoted by $\overline{u}\langle k[p] : G\rangle$ without ";" attached after it, is either with a local process *in parallel*, e.g. $P \mid \overline{a}\langle s[p] : G\rangle$ or floating around the network in the global queue (introduced in Section 5.4), e.g. $H \cdot \overline{a}\langle s[p] : G\rangle$. Note that, an invitation is different from an interaction message, which queues in either local or global queue; an invitation only queues in the global queue. Invitations are delivered by applying rules [$P-$request b/f] or [$P-$accept b/f] along *shared channels* $(u, u', \ldots)$. When an invitation is sent out from a local process, it implies that the local process takes action of free/bound request; while an invitation enters a local process, it implies the local process takes action of free/bound accept.

As mentioned above, [$P-$accept b/f] *do not mean* a process has joined a session-role. An endpoint uses [$P-$join] to claim to join a session-role in a particular session. This syntax together with [$P-$request b/f] and [$P-$accept b/f] give the calculus the flexibility for realising delegation.

u	::=	$a, b \quad \mid \quad x, y$	$[P-\text{shared}]$
k	::=	$s, s' \quad \mid \quad x, y$	$[P-\text{session}]$
v	::=	$a \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{c}$	$[P-\text{value}]$
e	::=	$v \mid x \mid e + e' \mid e \cap e' \mid \neg e \mid \cdots$	$[P-\text{expression}]$
S	::=	$\mathsf{int} \mid \mathsf{string} \mid \mathsf{bool} \mid G \mid T \mid U$	$[P-\text{sort}]$
U	::=	$\mathtt{mode}(T[p])$	$[P-\text{Use}]$
$\mathtt{im}$	::=	$\mathtt{I} \mid \mathtt{IO}$	$[P-\text{I-mode}]$
$\mathtt{mode}$	::=	$\mathtt{im} \mid \mathtt{O}$	$[P-\text{I/O-mode}]$
P	::=	$\mathsf{new}\ a : \mathtt{im}(G[p]).P$	$[P-\text{n creation}]$
	$\mid$	$\overline{u}(k[p] : G).P$	$[P-\text{request b}]$
	$\mid$	$\overline{u}\langle k[p] : G\rangle; P$	$[P-\text{request f}]$
	$\mid$	$u(y[p] : G).P$	$[P-\text{accept b}]$
	$\mid$	$u\langle y[p] : G\rangle; P$	$[P-\text{accept f}]$
	$\mid$	$\mathtt{join}(s[p]).P$	$[P-\text{join}]$
	$\mid$	$\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q$	$[P-\text{conditional}]$
	$\mid$	$k[p, q]!l\langle e\rangle; P$	$[P-\text{selection}]$
	$\mid$	$k[p_1, p_2]?\{l_i(x_i).P_i\}_{i\in I}$	$[P-\text{branching}]$
	$\mid$	$P \mid Q$	$[P-\text{parallel}]$
	$\mid$	$\mathbf{0}$	$[P-\text{inact}]$
	$\mid$	$\mu X\langle e\rangle(x).P \quad \mid \quad X\langle e\rangle$	$[P-\text{recursion}]$
	$\mid$	P_{rt}	$[P-\text{runtime}]$
P_{rt}	::=	$(\nu s)P \mid (\nu a)P$	$[P-\text{hiding}]$
	$\mid$	$s[p] : h$	$[P-\text{queue}]$
h	::=	$\emptyset \mid m \cdot h$	$[P-\text{local queue}]$
m	::=	$s\langle p, q, l\langle v\rangle\rangle \ \mid \ \overline{a}\langle s[p] : G\rangle$	$[P-\text{cm-message/invitation}]$

Figure 5.3: The Syntax of Local Processes

Example 5.3.1 (accept, join, and delegation). Assume there is a web-service, who is able to play role *Server* in session s guided by protocol G. Assume the server needs to deal with many requests from clients concurrently, the server thus delegates these tasks to the trusted agents, i.e. its deputies, who can as well play role *Server* in protocol G. Assume process P_{server} has a list of the shared names of the trusted agents. Let *agent* be the variable of shared name of the trusted agent. Let P_{server} be

$$P' \mid \mu X\langle agent\rangle(u).\mathsf{if}\ agent \neq \emptyset\ \mathsf{then}\ a(y[Server] : G).\overline{u}\langle y[Server] : G\rangle; X\langle \text{selectAgent}()\rangle\ \mathsf{else}\ \mathbf{0}$$

recursively accepts a new session, say s, to replace y and then delegates session-role $s[Server]$ to agent *agent*, which is private to P_{server}, by replacing u with *agent*. A process P' may be in parallel with the recursion for assigning an initial value of *agent* to the recursion. Also, function selectAgent() selects an agent from the list, then put the selected one into the recursive body. We can specify the recursion ends when $X\langle\emptyset\rangle$. Every process for an agent should have the shape

$$P_{\text{agent}} = agent(y'[Server] : G).\mathsf{join}(y'[Server]).P''$$

which says that endpoint *agent* will join session-role *Server* specified under G, by applying $\mathtt{join}(y'[Server])$, when such an invitation happens.

Definition 5.3.2 (the set of indexes). Define I as the set of indexes.

$[P-\text{selection}]$ and $[P-\text{branching}]$ are for *session interactions*, which are communications through an established session k. We make the sender p_1 and receiver p_2 in session k explicit by the notation $k[p_1, p_2]$. $[P-\text{selection}]$ defines a process involving in session k for sending value v with chosen label l from p_1 to p_2. Dually, $[P-\text{branching}]$ defines a process ready to receive whatever label $l_i, i \in I = \{1, ..., n\}$ has been chosen and behaves as $P_j\{v/x_j\}$ after replacing every x_j in P_j with v if label l_j has been chosen. For brevity, write $k[p_1, p_2]?l(x).P$ or $k[p_1, p_2]?(x).P$ for a single branch, and omit trailing $\mathbf{0}$. Conditional, parallel composition and inaction are standard. The *recursion* $\mu X\langle e\rangle(x).P$ defines a recursion where X is the recursion body defined in P with parameter x which is initiated to e. $X\langle e\rangle$ is the corresponding recursion call.

Some arguments arise from the necessity of embedding protocol G in the process calculus. The reasons are as follows: (I) Ensuring *deadlock-free* of processes in a session since behaviours are safely guided by the specifications, and (II) providing system the information about the *structure* that a session obeys. Without knowing to which protocol a session obeys, system can hardly guard/monitor the interactions in this session.

Once a session is created, the runtime syntax P_{rt} (not accessible to programmers) is used. To model the asynchronous interactions between a local process and its external monitor during runtime, a local queue, $s[p] : h$, created when $\texttt{join}(s[p])$ takes place and designed particularly for the endpoint playing session-role $s[p]$, is used as a bridge between them to buffer incoming (resp. outgoing) messages for which the endpoint is going to receive from (resp. send to) the network. Although the global queue (defined in Section 5.4) can realise the asynchronous interactions among processes, it is not enough for realising the ones between a process and its external monitor. Through buffering, a monitor can efficiently guard the incoming (to the local processes) and outgoing (to the network) messages for more than one endpoint.

The set of bound names in P is denoted by $\mathsf{bn}(P)$, the set of free names in P is denoted by $\mathsf{fn}(P)$, and the set of names in P is denoted by $\mathsf{n}(P)$. We view a session-role, say $k[p]$, as a composite of a session name and a role. Note that, a role p, specified under a global protocol, represents a pre-defined capability of an endpoint; therefore, it is not a name but is there for pattern-matching of capability. Below we illustrate the set of names, bound names, and free names of each process:

Definition 5.3.3 (the set of names, free/bound names of a process). $\cup$ is standard union set operation in set theory.

1. For $P = \overline{a}(k[p] : G); P'$, $\mathsf{n}(P) = \{a, k\} \cup \mathsf{n}(P')$, $\mathsf{fn}(P) = \{a\} \cup \mathsf{fn}(P')$, and $\mathsf{bn}(P) = \{k\} \cup \mathsf{bn}(P')$.

2. For $P = \overline{a}\langle k[p] : G\rangle; P'$, $\mathsf{n}(P) = \{a, k\} \cup \mathsf{n}(P')$, $\mathsf{fn}(P) = \{a, k\} \cup \mathsf{fn}(P')$, and $\mathsf{bn}(P) = \mathsf{bn}(P')$.

3. For $P = a(k[p] : G).P'$, $\mathsf{n}(P) = \{a, k\} \cup \mathsf{n}(P')$, $\mathsf{fn}(P) = \{a\} \cup \mathsf{fn}(P')$, and $\mathsf{bn}(P) = \{k\} \cup \mathsf{bn}(P')$.

4. For $P = a\langle k[p] : G\rangle.P'$, $\mathsf{n}(P) = \{a, k\} \cup \mathsf{n}(P')$, $\mathsf{fn}(P) = \{a, k\} \cup \mathsf{fn}(P')$, and $\mathsf{bn}(P) = \mathsf{bn}(P')$.

5. For $P = k[p, q]!l\langle e\rangle; P'$, $\mathsf{n}(P) = \{k\} \cup \mathsf{n}(e) \cup \mathsf{n}(P')$, $\mathsf{fn}(P) = \{k\} \cup \mathsf{n}(e) \cup \mathsf{fn}(P')$, and $\mathsf{bn}(P) = \mathsf{bn}(P')$.

6. For $P = k[p, q]?l(x).P'$, $\mathsf{n}(P) = \{k, x\} \cup \mathsf{n}(P')$, $\mathsf{fn}(P) = \{k\} \cup \mathsf{fn}(P')$, and $\mathsf{bn}(P) = \mathsf{n}(x) \cup \mathsf{bn}(P')$.

7. For $P = \mathsf{new}\ a : \mathtt{im}(G[p]).P'$, $\mathsf{n}(P) = \{a\} \cup \mathsf{n}(P')$, $\mathsf{fn}(P) = \mathsf{fn}(P')$, and $\mathsf{bn}(P) = \{a\} \cup \mathsf{bn}(P')$.
8. For $P = \mathtt{join}(k[p]).P'$, $\mathsf{n}(P) = \{k\} \cup \mathsf{n}(P')$, $\mathsf{fn}(P) = \mathsf{fn}(P')$, and $\mathsf{bn}(P) = \{k\} \cup \mathsf{bn}(P')$.
9. For other processes, the names and free/bound names of them are standard.

Convention 5.3.4 (Barendregt convention). In every process (and in every sequence of actions), bound names are different from each other and different from free names.

Convention 5.3.5. We call a process *initial* when a process does not contain any free variable and a runtime syntax. An initial process is thus always closed because it has no free variables.

5.3.2 The Structural Congruence of Local Processes

For convenience, we define the prefix of a process as two kinds:

Definition 5.3.6. Define two kinds of prefixes as follows:

$$\begin{array}{lll} \alpha_1 & ::= & \overline{u}(k[p] : G) \mid u(k[p] : G) \mid k[p_1, p_2]?l(x) \mid \mathtt{join}(s[p]) \mid \mathsf{new}\ a : \mathtt{im}(G[p]) \\ \alpha_2 & ::= & \overline{u}\langle k[p] : G\rangle \mid u\langle k[p] : G\rangle \mid k[p_1, p_2]!l\langle e\rangle \end{array}$$

Definition 5.3.7 (the laws of congruence). Let $\mathcal{R}$ be an equivalence relation over process. Then $\mathcal{R}$ is said to be a congruence relation

- if it is preserved by all elementary contexts over open process; that is, if $P_1\ \mathcal{R}\ P_2$, then

$$\begin{array}{rcl} \alpha_1.P_1 & \mathcal{R} & \alpha_1.P_2 \\ \alpha_2; P_1 & \mathcal{R} & \alpha_2; P_2 \\ P_1 \mid Q & \mathcal{R} & P_2 \mid Q \\ Q \mid P_1 & \mathcal{R} & Q \mid P_2 \\ (\nu n)P_1 & \mathcal{R} & (\nu n)P_2 \end{array}$$

- and, if $P\ \mathcal{R}\ Q$, then $\forall x, v$, $P\{v/x\}\ \mathcal{R}\ Q\{v/x\}$.

Definition 5.3.8 (processes structural congruence). The structural congruence is the smallest binary relation $\equiv$, which

1. is a congruence, i.e. abides to the law defined in Definition 5.3.7, and
2. satifies the following axioms

   1 $P \equiv Q$ whenever P is an alpha-conversion of Q.

   2 $P \mid \mathbf{0} \equiv P$, $P_1 \mid P_2 \equiv P_2 \mid P_1$, $P_1 \mid (P_2 \mid P_3) \equiv (P_1 \mid P_2) \mid P_3$.

   3 $(\nu n)(P_1 \mid P_2) \equiv P_1 \mid (\nu n)P_2$ if $n \notin \mathsf{fn}(P_1)$, $(\nu n)P_1 \mid (\nu n)P_2 \equiv (\nu n)(P_1 \mid P_2)$.

   4 $(\nu n)\mathbf{0} \equiv \mathbf{0}$.

   5 $s[p] : h \cdot s\langle p_1, p, l\langle v\rangle\rangle \cdot s\langle p_2, p, l'\langle v'\rangle\rangle \cdot h' \equiv s[p] : h \cdot s\langle p_2, p, l'\langle v'\rangle\rangle \cdot s\langle p_1, p, l\langle v\rangle\rangle \cdot h'$ if $p_1 \neq p_2$.

   6 $s[p] : h \cdot s\langle p, p_1, l\langle v\rangle\rangle \cdot s\langle p, p_2, l'\langle v'\rangle\rangle \cdot h' \equiv s[p] : h \cdot s\langle p, p_2, l'\langle v'\rangle\rangle \cdot s\langle p, p_1, l\langle v\rangle\rangle \cdot h'$ if $p_1 \neq p_2$.

In the definition of structural congruence, besides the standard rules defined in (117), Rules 5 and 6 are the rules for rearranging messages. Rule 5 says that, when two messages are received from different senders, these two messages can be permuted; while rule 6 says that, when two messages are going to send to different receivers, these two messages can be permuted.

5.3.3 The Semantics of Local Processes

In this section, we introduced the labelled transition system (LTS) and reductions rules (the τ rules in LTS) of local asynchronous processes. Equation 5.1 is the *full* version of action labels. The LTS includes the following action labels, which are also used in Chapters 7 and 8. Labels consist of τ-action (i.e. τ), session bound/free request and accept (i.e. $\overline{a}(s[p] : G)/\overline{a}\langle s[p] : G\rangle$, $a(s[p] : G)/a\langle s[p] : G\rangle$), shared channel creation $\mathsf{new}\ a : \mathtt{im}(G[p])$, join session-role $s[p]$ and two actions for session interactions: selection and branching (i.e. $s[p_1, p_2]!l\langle v\rangle$, $s[p_1, p_2]?l(x)$). Request and selection (resp. accept and branching) are often collectively called *output* (resp. *input*). Note, when a process announces to *join* a session-role $s[p]$ through a join label, only at this moment, the session-role becomes *active* and the local queue $s[p] : \emptyset$ is generated.

In Chapters 8, the action labels will be simplified by taking off $\mathtt{join}(s[p])$ and $\mathsf{new}\ a : \mathtt{im}(G[p])$ to only focus on session interaction actions.

$$\begin{aligned}\ell \;::=\;& \tau \mid \overline{a}(s[p] : G) \mid \overline{a}\langle s[p] : G\rangle \mid a(s[p] : G) \mid a\langle s[p] : G\rangle \mid \mathsf{new}\ a : \mathtt{im}(G[p]) \\ &\mid \mathtt{join}(s[p]) \mid s[p_1, p_2]!l\langle v\rangle \mid s[p_1, p_2]?l(v) \end{aligned} \tag{5.1}$$

Below we illustrate the set of names, bound names, and free names of each action:

Definition 5.3.9 (the set of names, free/bound names of an action). $\cup$ is standard union set operation in set theory.

1. For $\ell = \overline{a}(k[p] : G)$, $\mathsf{n}(\ell) = \{a, k\}$, $\mathsf{fn}(\ell) = \{a\}$, and $\mathsf{bn}(\ell) = \{k\}$.
2. For $\ell = \overline{a}\langle k[p] : G\rangle$, $\mathsf{n}(\ell) = \{a, k\}$, $\mathsf{fn}(\ell) = \{a, k\}$, and $\mathsf{bn}(\ell) = \emptyset$.
3. For $\ell = a(k[p] : G)$, $\mathsf{n}(\ell) = \{a, k\}$, $\mathsf{fn}(\ell) = \{a\}$, and $\mathsf{bn}(\ell) = \{k\}$.
4. For $\ell = a\langle k[p] : G\rangle$, $\mathsf{n}(\ell) = \{a, k\}$, $\mathsf{fn}(\ell) = \{a, k\}$, and $\mathsf{bn}(\ell) = \emptyset$.
5. For $\ell = k[p, q]!l\langle e\rangle'$, $\mathsf{n}(\ell) = \{k\} \cup \mathsf{n}(e)$, $\mathsf{fn}(\ell) = \{k\} \cup \mathsf{n}(e)$, and $\mathsf{bn}(\ell) = \emptyset$.
6. For $\ell = k[p, q]?l(x)$, $\mathsf{n}(\ell) = \{k, x\}$, $\mathsf{fn}(\ell) = \{k\}$, and $\mathsf{bn}(\ell) = \mathsf{n}(x)$.
7. For $\ell = \mathsf{new}\ a : \mathtt{im}(G[p])$, $\mathsf{n}(\ell) = \{a\}$, $\mathsf{fn}(\ell) = \emptyset$, and $\mathsf{bn}(\ell) = \{a\}$.
8. For $\ell = \mathtt{join}(k[p])$, $\mathsf{n}(\ell) = \{k\}$, $\mathsf{fn}(\ell) = \emptyset$, and $\mathsf{bn}(\ell) = \{k\}$.

Note that, according to Convention 5.3.4 (i.e. Barendregt convention), in every sequence of actions, bound names are different from each other and different from free names.

Example 5.3.10. Consider a sequence of actions:

$$\ell_1 \cdot \ell_2 \cdot \ell_3 \cdot \ell_4 = a(k[p] : G) \cdot k[p, q]!l\langle v\rangle \cdot a(k[q] : G) \cdot k[q, r]!l'\langle v'\rangle$$

It is wrong because it does not follow Barendregt convention as $a(k[p] : G)$ and $a(k[q] : G)$ have common bound name k. It should be revised to:

$$\ell_1 \cdot \ell_2 \cdot \ell_3 \cdot \ell_4 = a(k[p] : G) \cdot k[p, q]!l\langle v\rangle \cdot a(k'[q] : G) \cdot k'[q, r]!l'\langle v'\rangle$$

Then we have ℓ_1 binds ℓ_2 and ℓ_3 binds ℓ_4.

Definition 5.3.11 (subject of an action). Define $\mathsf{sbj}(\ell)$ as the subject of ℓ:

1. If it is an action for session interaction, its subject is the endpoint, represented by a session-role, who fires the action. For example, $\mathsf{sbj}(s[p,q]!l\langle v\rangle) = s[p]$ and $\mathsf{sbj}(s[p,q]?l(v)) = s[q]$.
2. If it is an action for request or accept, its subject is the initial shared name. For example, $\mathsf{sbj}(\overline{a}\langle s[p] : G\rangle) = \mathsf{sbj}(\overline{a}(s[p] : G)) = a$ and $\mathsf{sbj}(a\langle s[p] : G\rangle) = \mathsf{sbj}(a(s[p] : G)) = a$.
3. $\mathsf{sbj}(\tau) = \emptyset$.
4. Others are not defined.

The definition of subject of actions for session interaction is an abstraction of the viewpoint of a local process who takes the action. Thus, for action $s[p,q]!l\langle v\rangle$, the endpoint that takes this action is the process playing the session-role $s[p]$; similarly, for action $s[p,q]?l(v)$, the endpoint takes this action is the process playing the session-role $s[q]$. For request and accept actions, a, as a shared point, fires the action; while $s[p]$ in a request or an accept is not counted as a subject because it is for pattern-matching of capability, and this pattern-matching check is only done when the content is delivered to the endpoint.

Definition 5.3.12. The evaluation of an expression, denoted by $e \downarrow v$, is always possible whenever the initial process is closed and the logic of the boolean expressions is decidable.

Figure 5.4 lists the LTS rules of local processes, including the τ action. Each output is performed in two steps: (I) a local action spawns the message (i.e. invitations remain local and interaction messages are inserted in a local queue, say $s[p]$) and (II) a visible action sends out the message. Inputs are dual. For the first five rules, the output and input at local endpoints are globally invisible: $\lfloor$Arq f, Aac b/f$\rfloor$ respectively spawn and receive an invitation along a shared name. $\lfloor$Sel$\rfloor$ puts a message in a local queue, after evaluating v from expression e; dually, $\lfloor$Bra$\rfloor$ gets a message from a local queue with label l_j, so that the j-th process P_j receives value v and becomes $P_j\{v/x_j\}$ after replacing every x_j in P with v. $\lfloor$If$\rfloor$ is standard. The rest models actions which can be observed globally: $\lfloor$New$\rfloor$ is for shared name creation. Through this action, the monitor learns this freshly created shared name is able to play role p with behaviour defined in protocol G. Note that, there are two ways for a process to get a shared name: (I) One

is when it creates a new name by applying rule ⌊New⌋. This process always hides the shared name and makes it private. (II) Another is, without loss of generality, a process announces the shared name it knows to the public so that other processes can access this name by having the 0-mode common shared name.

⌊Join⌋ is for an endpoint to join a session by activating capability $s[p]$ and creating the corresponding local queue. ⌊req-out b/f⌋ sends out an invitation, its duality ⌊acc-in b/f⌋ receives an invitation from the outside. Note that ⌊req-out b⌋ (resp. ⌊acc-in b⌋) simultaneously creates the session s and requests (resp. accepts) the session role $s[p_j]$ at the endpoint represented by the shared name a. ⌊sel-out⌋ (resp. ⌊bra-in⌋) sends (resp. receives) a message from (resp. into) its local queue. Rules ⌊res⌋, ⌊par⌋, and ⌊str⌋ are standard according to (117).

5.4 The Calculus of the Network

In this section, this thesis introduces the formalism, including syntax and semantics, for the network, which can be an un-monitored, monitored, or a mixed network, where processes asynchronously interact with each other. The calculus considers both the aspects of the marginal and the global of the network.

5.4.1 The Syntax of the Network

The syntax of the network is given in Figure 5.5, where every notation, representing a network component, is separated by "|".

A process may have more than one endpoint. During runtime, these endpoints are driven by this process. A process P, residing locally, is a part of the network. " $\|''$ is used to represent that each network component exists in parallel. Let N_s denote the *static* network, which does not include the global queue; and let N denote the *dynamic* network which contains all components (including local processes and the global queue with messages).

Definition 5.4.1. Let $\mathtt{ep}(\mathsf{N})$ be the set of endpoints residing in N. As $\mathsf{N} = P$, $\mathtt{ep}(P)$ is the set of endpoints in process P.

Definition 5.4.2 (network composability). Two networks $\mathsf{N}_1 \stackrel{\text{def}}{=} \mathsf{N}_{s1} \parallel H_1$ and $\mathsf{N}_2 \stackrel{\text{def}}{=} \mathsf{N}_{s2} \parallel H_2$ are composable if and only if

$$\overline{a}\langle s[p] : G\rangle; P \xrightarrow{\tau} \overline{a}\langle s[p] : G\rangle \mid P \qquad \lfloor \textsc{Arq f} \rfloor$$

$$\overline{a}\langle s[p] : G\rangle \mid a(y[p] : G).Q \xrightarrow{\tau} Q\{s/y\} \qquad \lfloor \textsc{Aac b} \rfloor$$

$$\overline{a}\langle s[p] : G\rangle \mid a\langle s[p] : G\rangle; Q \xrightarrow{\tau} Q \qquad \lfloor \textsc{Aac f} \rfloor$$

$$s[p, q]!l\langle e\rangle; P \;\mid\; s[p_1]{:}\, h \xrightarrow{\tau} P \mid\; s[p_1]{:}\, h \cdot s\langle p_1, p_2, l\langle v\rangle\rangle \qquad (e \downarrow v) \qquad \lfloor \textsc{sel} \rfloor$$

$$s[p_1, p_2]?\{l_i(x_i).P_i\}_{i\in I} \mid s[p_2]{:}\, s\langle p_1, p_2, l_j\langle v\rangle\rangle \cdot h \xrightarrow{\tau} P_j\{v/x_j\} \mid s[p_2]{:}\, h \qquad \lfloor \textsc{bra} \rfloor$$

$$\mathsf{if}\ b\ \mathsf{then}\ P_{\mathsf{true}}\ \mathsf{else}\ P_{\mathsf{false}} \xrightarrow{\tau} P_b \quad (b \in \{\mathsf{true}, \mathsf{false}\}) \qquad \lfloor \textsc{if} \rfloor$$

$$\mathsf{new}\ a : \mathtt{im}(G[p]).P \xrightarrow{\mathsf{new}\ a:\mathtt{im}(G[p])} (\nu a)(P) \qquad \lfloor \textsc{new} \rfloor$$

$$\mathtt{join}(s[p]).P \xrightarrow{\mathsf{join}(s[p])} P \mid s[p]{:}\, \emptyset \qquad \lfloor \textsc{join} \rfloor$$

$$\overline{a}(s[p] : G).P \xrightarrow{\overline{a}(s[p]:G)} (\nu s)(P) \qquad \lfloor \textsc{req-out b} \rfloor$$

$$\overline{a}\langle s[p] : G\rangle \xrightarrow{\overline{a}\langle s[p]:G\rangle} \mathbf{0} \qquad \lfloor \textsc{req-out f} \rfloor$$

$$\mathbf{0} \mid P \xrightarrow{a(s[p]:G)} (\nu s)(\overline{a}\langle s[p] : G\rangle \mid P) \qquad \lfloor \textsc{acc-in b} \rfloor$$

$$\mathbf{0} \mid P \xrightarrow{a\langle s[p]:G\rangle} \overline{a}\langle s[p] : G\rangle \mid P \qquad \lfloor \textsc{acc-in f} \rfloor$$

$$s[p_1]{:}\, s\langle p_1, p_2, l\langle v\rangle\rangle \cdot h \xrightarrow{s[p_1,p_2]!l\langle v\rangle} s[p_1]{:}\, h \qquad (p_1 \neq p_2) \qquad \lfloor \textsc{sel-out} \rfloor$$

$$s[p_2]{:}\, h \xrightarrow{s[p_1,p_2]?l(v)} s[p_2]{:}\, h \cdot s\langle p_1, p_2, l\langle v\rangle\rangle \qquad (p_1 \neq p_2) \qquad \lfloor \textsc{bra-in} \rfloor$$

$$\frac{P \xrightarrow{\ell} P' \quad \mathsf{fn}(Q) \cap \mathsf{bn}(\ell) = \emptyset}{P \mid Q \xrightarrow{\ell} P' \mid Q} \qquad \frac{P \xrightarrow{\ell} P' \quad \mathsf{n}(\ell) \notin \tilde{n}}{(\nu\tilde{n})P \xrightarrow{\ell} (\nu\tilde{n})P'} \qquad \frac{P \equiv Q \quad Q \xrightarrow{\ell} Q' \quad Q' \equiv P'}{P \xrightarrow{\ell} P'} \qquad \lfloor \textsc{par,res,str} \rfloor$$

Figure 5.4: The Labelled Transition System of *Asynchronous* Local Processes

$$\begin{array}{rcll}
\mathsf{N}_s & ::= & P \mid \mathsf{N}_{s1} \parallel \mathsf{N}_{s2} \mid \mathbf{0} \mid (\nu n)\mathsf{N}_s & n \in \{u, k\} \quad \text{static network} \\
\mathsf{N} & ::= & \mathsf{N}_s \mid H \mid \mathsf{N}_1 \parallel \mathsf{N}_2 \mid (\nu n)\mathsf{N} & n \in \{u, k\} \quad \text{dynamic network} \\
H & ::= & \emptyset \mid m{\cdot}H & \text{global queue} \\
m & ::= & s\langle p, q, l\langle v\rangle\rangle \mid \overline{a}\langle s[p] : G\rangle & \text{message}
\end{array}$$

Figure 5.5: The Syntax for the Network

1. $\mathsf{ep}(\mathsf{N}_{s1}) \cap \mathsf{ep}(\mathsf{N}_{s2}) = \emptyset$.

2. There is no common message existing in H_1 and H_2.

Note that, we define the composabilty of monitored network as same as the above.

Definition 5.4.3 (composition of global queues). For any $H_1 = m_1 \cdot m_2 \ldots m_n$ and $H_2 = m'_1 \cdot m'_2 \ldots m'_m$, the composition of two global queues is denoted by $H_1 \cdot H_2 = m_1 \cdot m_2 \ldots m_n \cdot m'_1 \cdot m'_2 \ldots m'_m$.

Definition 5.4.4 (parallel composition of $\mathsf{N}_1 \parallel \mathsf{N}_2$). Assume $\mathsf{N}_1 \stackrel{\text{def}}{=} \mathsf{N}_{s1} \parallel H_1$ and $\mathsf{N}_2 \stackrel{\text{def}}{=} \mathsf{N}_{s2} \parallel H_2$. When N_1 and N_2 are composable, write the parallel composition of N_1 and N_2 as $\mathsf{N}_1 \parallel \mathsf{N}_2 = \mathsf{N}_{s1} \parallel \mathsf{N}_{s2} \parallel H_1 \cdot H_2$.

The global queue H is an abstract global transport among all network participants i.e. all endpoints share one common global queue. H consists of a sequence of *messages* $m_1 \ldots m_n$, where a message has the shape of either $\overline{a}\langle s[p] : G\rangle$ or $s\langle p_1, p_2, l\langle v\rangle\rangle$, ranged over m, which represents an in-transit-message, i.e. the message which has been sent from some sender but has not yet been received by its corresponding receiver. Abstracting a global queue is important, especially for *asynchronous* interactions, because

1. it captures the *dynamics* of the network by carrying floating messages from one endpoint to another, and

2. it illustrates that floating messages are interleaving in different sessions.

Consider, during runtime, in a session, a sender sends a message to a receiver. As the message enters the network, under the asynchronous environment, the receiver may not get the message immediately due to asynchrony. Because some local processes may be untrusted, the receiver is expecting an input but does not know if the corresponding sender has sent out a message to her. Since the message has been sent out, there is no message left at the sender side, the only place that a global observer (if there is any) can check is the global queue. Under the asynchronous environment, a message for session-interaction purpose should always be at three places: the sender's local queue, the global queue, or the receiver's local queue. The global queue abstracts the asynchronous interactive behaviours. Note that, for asynchronous interactions, the order of messages in the network is generally not preserved, except the messages from

a common session or two common session-roles, e.g. $s\langle p_1, p_2, l\langle v\rangle\rangle \cdot s\langle p_1, p_2, l'\langle v'\rangle\rangle$ (as in a TCP connection).

For synchronous environments, a message can only be outputted when its receiver is ready to receive it; while the receiver is ready, the outputting and inputting of a message are done *synchronously*. Thus there is no need to have a global queue in the network.

5.4.2 The Structural Congruence of the Network

Definition 5.4.5 (the laws of the network congruence). Let $\cong_{\mathsf{N}}$ be an equivalence relation over N. Then $\cong_{\mathsf{N}}$ is said to be a network congruence if it is preserved by all elementary contexts; that is, if $\mathsf{N}_1 \cong_{\mathsf{N}} \mathsf{N}_2$, then

$$\begin{array}{rcl} \mathsf{N}_1 \parallel \mathsf{N} & \cong_{\mathsf{N}} & \mathsf{N}_2 \parallel \mathsf{N} \\ \mathsf{N} \parallel \mathsf{N}_1 & \cong_{\mathsf{N}} & \mathsf{N} \parallel \mathsf{N}_2 \\ (\nu n)\mathsf{N}_1 & \cong_{\mathsf{N}} & (\nu n)\mathsf{N}_2 \end{array}$$

Definition 5.4.6 (network structural congruence). The structural congruence over networks is the smallest binary relation $\equiv_{\mathsf{N}}$, which

1. is a $\cong_{\mathsf{N}}$, i.e. abides to the law defined in Definition 5.4.5, and
2. satifies the following axioms

   1 $\mathsf{N}_1 \equiv_{\mathsf{N}} \mathsf{N}_2$ if N_1 is an alpha-conversion of N_2.

   2 $\mathsf{N} \parallel \mathbf{0} \equiv_{\mathsf{N}} \mathsf{N}$, $\mathsf{N}_1 \parallel \mathsf{N}_2 \equiv_{\mathsf{N}} \mathsf{N}_2 \parallel \mathsf{N}_1$, $\mathsf{N}_1 \parallel (\mathsf{N}_2 \parallel \mathsf{N}_3) \equiv_{\mathsf{N}} (\mathsf{N}_1 \parallel \mathsf{N}_2) \parallel \mathsf{N}_3$.

   3 $(\nu n)(\mathsf{N}_1 \parallel \mathsf{N}_2) \equiv_{\mathsf{N}} \mathsf{N}_1 \parallel (\nu n)\mathsf{N}_2$ if $n \notin \mathsf{fn}(\mathsf{N}_1)$.

   4 $(\nu n)\mathbf{0} \equiv_{\mathsf{N}} \mathbf{0}$.

   5 $H \cdot s\langle p, q, l\langle v\rangle\rangle \cdot s'\langle p', q', l'\langle v'\rangle\rangle \cdot H' \equiv_{\mathsf{N}} H \cdot s'\langle p', q', l'\langle v'\rangle\rangle \cdot s\langle p, q, l\langle v\rangle\rangle \cdot H'$ if $s \neq s'$ or $p \neq p'$ or $q \neq q'$.

In the definition of structural congruence of the network, rule 5 is the rule for rearranging messages. It says, globally, as long as two adjacent messages belong to different sessions, or they have different senders or receivers, they can be permuted.

5.4.3 The Semantics of the Network

Based on the labels defined in Equation 5.1, let $\ell^{\mathtt{O}}$ be the output action labels, $\ell^{\mathtt{I}}$ be the input action labels.

$$\begin{aligned} \ell^{\mathtt{O}} &::= \overline{a}(s[p] : G) \mid \overline{a}\langle s[p] : G\rangle \mid s[p_1, p_2]!l\langle v\rangle \\ \ell^{\mathtt{I}} &::= a(s[p] : G) \mid a\langle s[p] : G\rangle \mid s[p_1, p_2]?l(v) \end{aligned} \tag{5.2}$$

To have a general definition of network reduction rules, several mapping functions are defined below to make the description of reduction rules simple. Define msg as a function mapping an action to the message which it has outputted or is going to input:

$$\begin{aligned} \mathsf{msg} ::= {} & \overline{a}(s[p] : G) \mapsto \overline{a}\langle s[p] : G\rangle \mid \overline{a}\langle s[p] : G\rangle \mapsto \overline{a}\langle s[p] : G\rangle \\ \mid {} & a(s[p] : G) \mapsto \overline{a}\langle s[p] : G\rangle \mid a\langle s[p] : G\rangle \mapsto \overline{a}\langle s[p] : G\rangle \\ \mid {} & s[p_1, p_2]!l\langle v\rangle \mapsto s\langle p_1, p_2, l\langle v\rangle\rangle \mid s[p_1, p_2]?l(v) \mapsto s\langle p_1, p_2, l\langle v\rangle\rangle \end{aligned} \tag{5.3}$$

other actions are not defined. Note that as $\overline{a}\langle s[p] : G\rangle$ is viewed as a message, it is an invitation. Also, define dest as a function mapping an action, if it is for input and output, to this action's destination as:

$$\begin{aligned} \mathsf{dest} ::= {} & \overline{a}(s[p] : G) \mapsto a \mid \overline{a}\langle s[p] : G\rangle \mapsto a \\ \mid {} & a(s[p] : G) \mapsto a \mid a\langle s[p] : G\rangle \mapsto a \\ \mid {} & s[p_1, p_2]!l\langle v\rangle \mapsto s[p_2] \mid s[p_1, p_2]?l(v) \mapsto s[p_2] \end{aligned} \tag{5.4}$$

other actions are not defined.

The operational semantics of the network is a duality of the semantics of local processes. The actions happening at local processes are *marginal* actions to the network. From the global viewpoint, those marginal actions are *invisible*, and are treated as silent actions (τ), which can be viewed as reductions happening in the network. On the other hand, the messages sent *across* network can be globally observed. For example, as a message is sent from one local endpoint, say Alice, to another, say Bob, for whom are in different local domains, this message is outputted from Alice to the global queue, then absorbed (inputted) by Bob from the global queue. When a message is outputted into

$$\frac{P \xrightarrow{\ell^{\mathtt{O}}} P'}{P \parallel H \longrightarrow P' \parallel H \cdot \mathsf{msg}(\ell^{\mathtt{O}})} \quad \lfloor \text{N-OUT} \rfloor$$

$$\frac{P \xrightarrow{\ell^{\mathtt{I}}} P' \qquad \mathsf{dest}(\ell^{\mathtt{I}}) \in \mathsf{ep}(P)}{P \parallel \mathsf{msg}(\ell^{\mathtt{I}}) \cdot H \longrightarrow P' \parallel H} \quad \lfloor \text{N-IN} \rfloor$$

$$\frac{P \xrightarrow{\mathsf{new}\ a:\mathtt{im}(G[p])} (\nu a)(P')}{P \parallel H \longrightarrow (\nu a)(P') \parallel H} \quad \lfloor \text{N-NEW} \rfloor$$

$$\frac{\mathtt{join}(s[p]).P \xrightarrow{\mathsf{join}(s[p])} P \mid s[p]{:}\emptyset}{\mathtt{join}(s[p]).P \parallel H \longrightarrow P \mid s[p]{:}\emptyset \parallel H} \quad \lfloor \text{N-JOIN} \rfloor$$

$$\frac{\mathsf{N}_1 \longrightarrow \mathsf{N}_1'}{\mathsf{N}_1 \parallel \mathsf{N}_2 \longrightarrow \mathsf{N}_1' \parallel \mathsf{N}_2} \quad \lfloor \text{N-PAR} \rfloor$$

$$\frac{\mathsf{N} \longrightarrow \mathsf{N}'}{(\nu \tilde{n})\mathsf{N} \longrightarrow (\nu \tilde{n})\mathsf{N}'} \quad \lfloor \text{N-RES} \rfloor$$

$$\frac{\mathsf{N} \equiv_{\mathsf{N}} \mathsf{N}_0 \quad \mathsf{N}_0 \longrightarrow \mathsf{N}_0' \quad \mathsf{N}_0' \equiv_{\mathsf{N}} \mathsf{N}'}{\mathsf{N} \longrightarrow \mathsf{N}'} \quad \lfloor \text{N-STR} \rfloor$$

Figure 5.6: The Reduction Rules of the Network

the global queue, we say this message, from the global viewpoint, *enters* the network; then this message can be observed through watching the global queue. Conversely, when a message is inputted by an endpoint, we say this message leaves the network.

Figure 5.6 shows the reduction (τ actions) rules for the network. Remember, the rules are based on the global viewpoint. Rule $\lfloor$N-OUT$\rfloor$ says, when a local process P outputs a message with action label $\ell^{\mathtt{O}}$, globally it is a marginal invisible action putting the message (corresponding to $\ell^{\mathtt{O}}$) into global queue H. Rule $\lfloor$N-IN$\rfloor$ is the duality of $\lfloor$N-OUT$\rfloor$. A local process P inputs a message with action label $\ell^{\mathtt{I}}$, which can be finally absorbed by P since $\mathsf{dest}(\ell^{\mathtt{I}})$ is in $\mathsf{ep}(P)$. Globally, it is a marginal invisible action absorbing the message (corresponding to $\ell^{\mathtt{I}}$) from global queue H. Rule $\lfloor$N-NEW$\rfloor$ represents that newing a shared name happens locally. Since we restrict that an `im`-mode shared name is unique in the network, to satisfy this restriction, in practice and in monitoring, one can assume that there are distributed routing tables in the global queue, H, for recording every created name. Rule $\lfloor$N-JOIN$\rfloor$ says joining a session-role is locally decided and it will inform the routing mechanism if there is any. Rule $\lfloor$N-PAR$\rfloor$ says a reduction of a part of the network does not affect other parts in the network.

Rule $\lfloor$N-RES$\rfloor$ says that if a network N can be reduced to N', the bound network $(\nu n)\mathsf{N}$ can be reduced to $(\nu n)\mathsf{N}'$. Rule $\lfloor$N-STR$\rfloor$ is standard.

Figure 5.7 illustrates the rules of LTS of the network, represented by the *observable* global transport. Note that, $\xrightarrow{\ell}_{\mathrm{g}}$ is particularly used to denote the transitions which are *globally observable* through observing the global queue. $\mathsf{N} \xrightarrow{\ell}_{\mathrm{g}} \mathsf{N}'$ represents the global observability of the network. By watching the transitions happening in the global queue, where actions are visible via inputting and outputting messages, the rules are as the dual of the reduction rules, which generalise the LTS of local processes. Rules {REQ-B/F} and {SEL} are summarised as the general output rule {OUT}, which is the dual of reduction rule $\lfloor$N-IN$\rfloor$; while rules {ACC-B/F} and {BRA} are summarised as the general input rule {IN}, which is the dual of reduction rule $\lfloor$N-OUT$\rfloor$. Note that, $H/\mathsf{msg}(\ell)$ means taking off message $\mathsf{msg}(\ell)$ from the set of messages in H. Rule {REQ-B} says that, as action $\overline{a}(s[p] : G)$ takes place, it implies that there exists a local process in the network taking this action for newing a session, and rule $\lfloor$REQ-OUT B$\rfloor$ defined in Figure 5.4 is applied. As a message leaves the global queue, there should be exist local process receiving it as an input. Rule {OUT} means there is a local process in N_s outputting a message to the global queue, thus the destination of action ℓ should not be inside N_s but outside it. Rule {IN} means local process P is going to receive a message from the global queue, thus N_s does not change its configuration. Rule {TAU} summarise the reduction rules defined in Figure 5.6. Rule {RES} and {STR} are standard according to (117). Rule {PAR} says that, the bound names of action ℓ should not be any free name appearig in network N_2, and it should not be absorbed by any process in network N_2 (i.e. its destination is not in N_2). The global transport in the network is used to abstract the network dynamics. The LTS of the network (Figure 5.7), which describes the observable actions via the LTS of global transport, and the LTS of local processes (Figure 5.4) together illustrate that the actions happening in the global queue and those happening in the endpoints are dual.

Remark 5.4.7. The actions happening in local processes are the dualities of those happening in the network, and vice versa.

5.4.4 Well-formedness

A network N which satisfies the following conditions is called *well-formed*: (I) it contains at most one global queue. (II) Two local processes never have the same shared name

$$\{\textsc{Req-b}\} \quad H \xrightarrow{\overline{a}(s[p]:G)}_{\mathrm{g}} H \cdot \overline{a}\langle s[p] : G\rangle$$

$$\{\textsc{Req-f}\} \quad H \xrightarrow{\overline{a}\langle s[p]:G\rangle}_{\mathrm{g}} H \cdot \overline{a}\langle s[p] : G\rangle$$

$$\{\textsc{Acc-b}\} \quad \overline{a}\langle s[p] : G\rangle \cdot H \xrightarrow{a(s[p]:G)}_{\mathrm{g}} H$$

$$\{\textsc{Acc-f}\} \quad \overline{a}\langle s[p] : G\rangle \cdot H \xrightarrow{a\langle s[p]:G\rangle}_{\mathrm{g}} H$$

$$\{\textsc{Sel}\} \quad H \xrightarrow{s[p_1,p_2]!l\langle v\rangle}_{\mathrm{g}} H \cdot\ s\langle p_1, p_2, l\langle v\rangle\rangle$$

$$\{\textsc{Bra}\} \quad s\langle p_1, p_2, l\langle v\rangle\rangle \cdot H \xrightarrow{s[p_1,p_2]?l(v)}_{\mathrm{g}} H$$

$$\{\textsc{Out}\} \quad \frac{H \xrightarrow{\ell}_{\mathrm{g}} H \cdot \mathsf{msg}(\ell) \quad \ell \text{ is output} \quad P \xrightarrow{\ell} P' \quad \mathsf{dest}(\ell) \notin \mathsf{ep}(\mathsf{N}_s)}{P \parallel \mathsf{N}_s \parallel H \xrightarrow{\ell}_{\mathrm{g}} P' \parallel \mathsf{N}_s \parallel H \cdot \mathsf{msg}(\ell)}$$

$$\{\textsc{In}\} \quad \frac{H \xrightarrow{\ell}_{\mathrm{g}} H/\mathsf{msg}(\ell) \quad \ell \text{ is input} \quad P \xrightarrow{\ell} P'}{P \parallel \mathsf{N}_s \parallel H \xrightarrow{\ell}_{\mathrm{g}} P' \parallel \mathsf{N}_s \parallel H/\mathsf{msg}(\ell)}$$

$$\{\textsc{tau, res}\} \quad \frac{\mathsf{N}_s = P \parallel \mathsf{N}_{s0} \quad \mathsf{N}_s{}' = P' \parallel \mathsf{N}_{s0} \quad P \xrightarrow{\ell} P'}{\mathsf{N}_s \xrightarrow{\tau}_{\mathrm{g}} \mathsf{N}_s{}'} \qquad \frac{\mathsf{N} \xrightarrow{\ell}_{\mathrm{g}} \mathsf{N}' \quad \mathsf{n}(\ell) \notin \tilde{n}}{(\nu\tilde{n})\mathsf{N} \xrightarrow{\ell}_{\mathrm{g}} (\nu\tilde{n})\mathsf{N}'}$$

$$\{\textsc{str}\} \quad \frac{\mathsf{N} \equiv \mathsf{N}_0 \xrightarrow{\ell}_{\mathrm{g}} \mathsf{N}_0' \equiv \mathsf{N}'}{\mathsf{N} \xrightarrow{\ell}_{\mathrm{g}} \mathsf{N}'} \qquad \{\textsc{par}\} \frac{\mathsf{N}_1 \xrightarrow{\ell}_{\mathrm{g}} \mathsf{N}_1' \quad \mathsf{bn}(\ell) \cap \mathsf{fn}(\mathsf{N}_2) = \emptyset \quad \mathsf{dest}(\ell) \notin \mathsf{ep}(\mathsf{N}_2)}{\mathsf{N}_1 \parallel \mathsf{N}_2 \xrightarrow{\ell}_{\mathrm{g}} \mathsf{N}_1' \parallel \mathsf{N}_2}$$

Figure 5.7: The Labelled Transition System of the Global Transport

with *input* mode (i.e. `I` or `IO`). We write $\prod_i P_i$ to denote $P_1 \parallel P_2 \cdots \parallel P_n$. Note that well-formedness is preserved by reduction.

As for (II), it means that, let *mode* be a function mapping a shared name to its I/O mode whenever $a \in \mathsf{n}(P)$ and $a \in \mathsf{n}(Q)$ where $P \neq Q$, if $a \in \mathsf{n}(P)$, $mode(a)$=`I` or $mode(a)$=`IO`, then for any $a \in \mathsf{n}(Q)$, its mode should be `O` (for output only). This restriction comes from the setting that there is only one *input* mode for a common shared channel in the network; a shared channel can be used by several endpoints as an *output* channel to communicate to one and only one target, who has the same shared channel with *input* mode. If N does not contain a global queue and has no hiding, then N is *static*, i.e. N_s, because it has no dynamic. A network is called *initial* if the local processes do not contain operator for new name hiding and if the message queue H is empty.

5.4.5 Global Queue v.s. Local Queue

A "queue", which is in between processes, is there for abstracting asynchronous behaviours through buffering messages which cannot be immediately absorbed by an expected receiver. In the proposed calculus, we design both global queue, which is unique in the network, and local queues, which belong to specific endpoints, to realise asynchronous interactions. If the purpose is only for realising asynchronous processes, we only need one of them. But for *asynchronous* monitoring, we need both of them: Use the global queue to represent asynchronous interactions among concurrent processes, and use local queues to capture asynchronous interactions between an external monitor and its, more than one, corresponding local processes. Having local queues in the monitoring framework also makes our model closer to the reality: in real-life systems, local processes generally use buffers to manage interaction messages. When we have local queues, having a global queue is to make interleaving messages, which corresponding to interleaving actions in different sessions, become explicit and *observable* globally.

5.5 The Functionalities of the Capability-passing π-Calculus

5.5.1 Mobility

As this thesis claims that the capability-passing π-calculus is based on the asynchronous π-calculus, thus it should be able to represent not only concurrent interactions but also mobility. Note that, in the π-calculus,

$$\overline{x}\langle y\rangle \mid x(z).\overline{z}\langle 10\rangle \longrightarrow \overline{y}\langle 10\rangle,$$

the mobility is realised by name passing shown above; although the capability-passing π-calculus has the similar name-passing mechanism shown below, the mobility is not done by replacing variable u by the received name a, but is done by passing capability $s[q_2]$ to a through $\overline{a}\langle s[q_2] : G\rangle$:

$$s[p, q_1]!\langle a\rangle; Q' \parallel s[p, q_1]?(u).\overline{u}\langle s[q_2] : G\rangle; P'$$

in which a process playing session-role $s[q_1]$ receives the knowledge about where shared point a is, and then passes the endpoint represented by a the capability for playing session-role $s[q_2]$ specified under G.

The mobility will be explained by using the scenario of "mobile phones" example for explaining how the π-calculus is powerful enough to realise mobility.

The follows illustrate some key points which make the representation of mobility of the capability-passing π-calculus different from the original asynchronous π-calculus:

1. Every shared channel at every endpoint is specifically I/O typed with mode `I`, `IO`, or `O` to declare their I/O capability.

2. When a name, say a, is transmitted by a session interaction action, say $s[p, q]!l\langle a\rangle$, from endpoint playing $s[p]$ to endpoint playing $s[q]$, it is viewed as a propagation rather than name passing (introduced in (117)) such that the endpoint playing $s[p]$ still *knows* a through either of the following ways:

   (a) If a is with input mode, `I` or `IO`, at the endpoint playing $s[p]$, then this endpoint still binds a with mode `I` or `IO`. a with mode `I` or `IO` is alive since it is created by a process (i.e. an endpoint), and it will die until this process terminates.

(b) If a is with output mode, `O` or `IO`, then this endpoint still has the knowledge about a with mode `O` or `IO`, which means the endpoint knows where to communicate with a.

And, the endpoint playing $s[q]$ has the knowledge about a with mode `O` because action $s[p,q]!\langle a\rangle$ passes a to the endpoint playing $s[q]$.

3. By applying $[P-$request b/f$]$, an endpoint loses a session-role $s[p]$ by delegating it to another endpoint, i.e. by requesting another endpoint to play this session-role. If the endpoint is represented by a specific shared name b, then the delegation is

$$\overline{b}\langle s[p] : G\rangle$$

which means giving up for playing $s[p]$ and requesting shared chanel b to bind $s[p]$ for playing this role.

Although it seems some restrictions are attached to this calculus, this session-based calculus can realise mobility with a high-level language. The following section with a whole example illustrates this point.

In (117), it uses example "mobile phones" to demostrate the mobility that the π-calculus provides. Its scenario is described as follows: Assume a user uses mobile phone in a car. When the car runs from one area to another, say from area A to area B, the mobile phone transmitter of area A will be correspondingly changed to the one of area B. This is controlled by a control center, which is responsible for switching the mobile phone's connection of the transmitter in area A to the connection of the transmitter in area B.

From this example, we learn that the mobility is represented by the *ability of possessing a channel*. When a process possesses or knows a channel, it has the *ability to use it for sending or receiving* messages. Recall that, based on our assumptions (see Chapter 4), there are two kinds of names (or channels), which are session channel and shared channel. A session channel is designed for interactions, while a shared channel is viewed as a shared point linking connections to a specific endpoint. Particularly, a session-role inside a session channel is as important because it is a kind of *name*, which represents the capability of interactive behaviours. With the similar idea, but different representation, the capability-passing π-calculus mainly applies *session-roles passing*

to show the change of *possessing a capability* from one process to another process to represent the mobility.

Here we use the capability-passing π-calculus to represent the mobility of example "mobile phones". In the session-based calculus, every process is viewed as a session-role involving in a session. In this example, there are three session-roles: Role *car* as a car, role *trans* as a transmitter, and role *control* as a control center. Once a session is established, the interactions are done by passing messages among the session-roles. Even though the car will move from one area, covered by a transmitter, to another area, covered by another transmitter, only the shared channel of the transmitter will change, but the session-role *trans* is still the same.

Let P_{trans_i} be the process for a transmitter i, using shared name a_i for playing role *trans*, specified under protocol G, in session y_i, which is a session variable for P_{trans_i}:

$$P_{\text{trans}_i} \quad = \quad a_i(y_i[trans] : G).\mu X.y_i[control, trans]?\{\text{talk}().P_{\text{talkcar}}.X, \text{lose}(tn).P_{\text{Idtrans}}\}$$

where $i = \{1, 2\}$, and tn is the variable of name for a transimitter. A transmitter is recursively waiting for the command from *control*. If the command from *control* is "talk" i.e. branch "talk" has been chosen, then it executes P_{talkcar} and then keeps listening. If the command from *control* is "lose(tn)", then it executes P_{Idtrans}. P_{Idtrans} is a process to idle the transmitter P_{trans_i} by removing the capability of P_{trans_i} of playing role $s[trans]$, and, at the same time, requests shared channel tn, informed from *control*, to play session-role $s[trans]$.

$$P_{\text{Idtrans}} = \overline{tn}\langle y_i[trans] : G\rangle; P_{\text{trans}_j} \longrightarrow \overline{tn}\langle y_i[trans] : G\rangle \;\mid\; P_{\text{trans}_j}$$

where $j \in \{1, 2\}$ and $i \neq j$. There can be many P_{trans_j}, other transmitters, positioning in parallel after the request $\overline{tn}\langle y_i[trans] : G\rangle$. If the *control* assigns the shared channel tn correctly, there must exist some P_{trans_j} having a shared channel that matches tn, i.e. $tn = a_j$ so that it can accept this request.

Note that, with the capability-passing π-calculus, there is no need to inform the process who plays role *car*, as what the "mobile phones" example in (117) did. It is because the process playing *car* only needs to know how to communicate with other session-roles. Every session-role is bound to some shared channel during communications, and these bindings are decided by the actions of request and accept. When action

$\overline{tn}\langle s[trans] : G\rangle$ in P_{trans_i} takes place, it means the shared channel a_i in P_{trans_i} for binding sesion-role $s[trans]$ has given up the capability of playing role $s[trans]$; when some P_{trans_j} accepts this request, its shared name a_j will take this capability by binding to the session-role $s[trans]$.

If there is a routing table for routing messages among session-roles, the routing table will change the record from $s[trans] \mapsto a_i$ to $s[trans] \mapsto a_j$. It implies that, the session-role $s[car]$ will not feel there is a change because its communications with $s[trans]$ are still there even though the shared name of $s[trans]$ is changed from a_i to a_j.

To complete this example, let process P_{talkcar} be the process to communicate with process P_{car}, which plays role car in the session:

$$\begin{array}{rcl} P_{\text{talkcar}} & = & \mathsf{if}\ info = \mathsf{true}\ \mathsf{then}\ y_i[trans, car]!\text{info}\langle info\rangle\ \mathsf{else}\ y_i[trans, car]!\text{error}\langle errMsg\rangle \\ P_{\text{car}} & = & c(y_{\text{car}}[car] : G).\mu X.y_{\text{car}}[trans, car]? \\ & & \quad \{\text{info}(x).y_{\text{car}}[car, trans]]!\text{reply}\langle\text{message}_1\rangle, \\ & & \quad\ \text{error}(y).y_{\text{car}}[car, trans]!\text{report}\langle\text{message}_2\rangle\} \end{array}$$

P_{talkcar} is a process to communicate with role car. It can be any kind of interactions with car. Here it assumes that when there is a new information, which is denoted by $info = \mathsf{true}$, role $trans$ sends car the coming information. Similarly, when there is an error happening, role $trans$ sends car the error message. As for P_{car}, it firstly accepts to join a session to play role car, specified under protocol G. Then it recursively waits for messages from $trans$. If it is for "info", it replies $trans$ with message_1. If it is for 'error", it reports to $trans$ with message_2.

The process of control center, playing role $s[control]$, is described below:

$$P_{\text{control}} = \mathsf{if}\ sig > \text{bottle}\ \mathsf{then}\ s[control, trans]!\text{talk}\langle\rangle\ \mathsf{else}\ s[control, trans]!\text{lose}\langle tn\rangle$$

It assumes that when the signal, denoted by sig, is strong enough, represented by $sig >$ bottle, $control$ asks role $trans$ to continue using the same channel to communicate with car by selecting branch "talk"; otherwise, it selects branch "lose" to ask it to lose the current channel and to grab another channel, tn, to continue playing role $trans$ for communicating with car.

5.5.2 Pattern-matching

Primitives $\overline{u}(k[p] : G)$ (or $\overline{u}\langle k[p] : G\rangle$) and $u(y[p] : G)$ (or $u\langle y[p] : G\rangle$) compose a pair for *pattern-matching*. Pattern-matching mechanism is a kind of access control through identifying endpoints for playing each session-role asynchronously and linearly one by one. When a session is created through linear invitations, *linearity* is automatically satisfied, which means there is no conflict (racing) at session-roles. When an invitation comes from session k, which is *fresh* to the recipient, primitive $u(y[p] : G).P$ is used to match this request with u, p, and G. If it is matched, this request is accepted by replacing variable y with session k. When k is *not fresh* to the recipient, the primitive $u\langle y[p] : G\rangle.P$ is used for pattern-matching. Note that, since y should has been replaced by some session, say s, the invitation should match not only u, p, and G, but also s.

Example 5.5.1 (access control through pattern-matching). Assume a process P, defined as below,

$$P = \overline{a}(s[p_1] : G) \mid \overline{c}\langle s[p_2] : G\rangle$$

wants to invite endpoints a and c for playing roles p_1 and p_2 in session s, specified under G. Assume there are two processes P_{rev1} and P_{rev2},

$$\begin{aligned} P_{\text{rev1}} &= a(y[p_1] : G).P'_{\text{rev1}} \\ P_{\text{rev2}} &= c(y[p_3] : G).P'_{\text{rev2}} \end{aligned}$$

then only P_{rev1} can accept P's invitation for playing role p_1 specified under G because P_{rev2} is only eligible for playing role p_3 but not p_2.

Another example is, assume, for the same P, there are two processes P_{rev3} and P_{rev4},

$$\begin{aligned} P_{\text{rev3}} &= a(y[p_1] : G).b\langle y[q] : G\rangle \\ P_{\text{rev4}} &= a(y[p_1] : G').b\langle y[q] : G'\rangle \end{aligned}$$

both of them can play role p_1 but defined under different protocols G and G'. Then only P_{rev3} matches the pattern to accept the invitation from P. Note that, in P_{rev3}, when invitation $\overline{a}\langle s[p_1] : G\rangle$ matches $a(y[p_1] : G)$, the process of recipient becomes

$$b\langle y[q] : G\rangle\{s/y\} = b\langle s[q] : G\rangle$$

so that the next invitation for endpoint b should match b, q, G and s.

Access control is also realised through runtime monitoring where monitors have specifications with logical predicate.

Example 5.5.2 (pattern-matching for preventing information leakage). Assume a network has the following local processes in parallel:

$$(\nu s)(\overline{a}\langle s[p] : G\rangle; P) \parallel H \parallel a(y[p] : G).Q \parallel b(y'[p] : G).Q'$$

where $a(y[p] : G).Q$ and $b(y'[p] : G).Q'$ are two local processes in the network. As $a \neq b$, the process of $b(y'[p] : G).Q'$ cannot match the invitation $\overline{a}\langle s[p] : G\rangle$ to get it. Moreover, when monitoring is on, $a = b$ is not allowed because, in the network, there is one and only one endpoint having a with input mode (based on the assumptions defined in Chapter 4).

5.5.3 Propagation

As a shared name, say a, is passed from one endpoint to another, the endpoint (driven by a process) who sends it will not lose a and the endpoint who receives it will only get a with output mode, which means it only has the knowledge about a but cannot use a to play any session-role. The modes of input and output distinguish the capability of a shared channel, and therefore distinguish the capability of a process who has the channel. When a process has a shared channel with input mode, it can use it *to bind a session-role* for playing this role because it can listen (i.e. input) the requests from the outside; while a process has the same shared channel but with `O`-mode (i.e. not including `IO`-mode), it can only send (i.e. output) messages through this channel with her knowledge about this channel.

Thus passing a shared name in our calculus is different from the standard name-passing mechanism in which, when a process passes a name to another, the sender loses the name, while the receiver gains it. For this difference, we say passing a shared name as a *propagation*: to propagate the knowledge about a shared name.

The propagation is realised by the following mechanism, which is controlled by the change of configurations of specifications (introduced in Chapter 6):

1. When a process *creates* a new name, say a, the specification guarding this process will add a with *input* mode. It means that whenever a process creates a new name, the process has this name for input.

2. When a specification guarding a process (i.e. an endpoint) has a shared name, say a, this process can send out name a.

3. When a process receives a name, say a, from another process, the specification for this process will add a with output mode.

As it comes to the calculus of monitoring, every system monitor possesses local specifications, which are projected from the global ones, of the endpoints which she monitors.

6

Specifications for Governance

Overview This chapter, as a preparation for Chapter 7, formally introduces the specifications for session-based dynamic monitoring. These specifications are built upon standard *MPST* and *MPSA* technologies which bring essential mechanisms such as the global specifications for network-wise interactions, the local specifications for endpoint processes's behaviours, and endpoint projection from a global specification. Section 6.1 gives two motivating examples. One motivates the effectiveness of session-based specifications, another motivates the desire of permutation mechanism of specifications. Section 6.2 introduces the grammar of global session-based specifications, and Section 6.3 states the consistency principles, together with well-formedness and well-assertedness properties, to validate global specifications. Section 6.4 gives the grammar of local session-based specifications, and Section 6.5 introduces projection mechanism for projecting a global specification to endpoint roles with local specifications. The permutation mechanism is introduced in Section 6.6, which is designed to support a monitor for determining messages passing in asynchronous monitoring environment.

6.1 Motivating Example

This section provides two motivating examples. One gives an illustration that a session-type based specification enjoys conciseness and expressiveness for regulating concurrent interactions. Another shows that, since asynchrony leads to non-deterministic results,

permutation mechanism, introduced in Section 6.6, is designed for specifying asynchronous interactions.

6.1.1 The Effectiveness of a Session-based Specification

Assume an external monitor, guarding the local process representing Principal A, has the following concurrent tasks:

(I) Principal A sends Principal B a request for buying a book with a content "buy War and Peace". Finish.

(II) Principal A sends Principal C a request for booking a book with a content "book Harry Potter". Finish.

(III) Principal A gets a feedback as "231342". Finish.

(IV) Principal A gets a feedback as "B231342". Finish.

Note that (I) and (II) (resp. (III) and (IV)) may happen concurrently. When the structure of *sessions* is not applied, at the conversation-level, the monitor watching conversations can not realise if content of "231342" is coming from Principal B or C, neither can she ensure the content of "B231342". Note that, as we say it is at the conversation-level, or the session-level, we mean that the messages have not yet entered the domain of the receiver's location. If a message arrives at the endpoint, the endpoint may ask a key from the message to check if the message is valid. The session-type-based specifications are for the conversation-level messages when those messages have not yet arrived at the receivers. Even if Principal A is informed that "B231342" is a feedback from Principal B because the message claims so, she does not know whether the format of the content is correct. In practice, before an interaction starts, they firstly establish a commitment through negotiation to specify the format of transimitted data. It can be done by a negotiation protocol, which in fact can apply the structure of sessions.

With the structure of sessions, a protocol can be easily done with session-based specifications which have specified every step of interactions. The following protocols are represented by session types:

$$\begin{array}{lrcl} \text{Buy Protocol} & G_{buy} & = & A \to B : (\mathsf{string}).B \to A : (\mathsf{formatB}).\mathsf{end} \\ \text{Book Protocol} & G_{book} & = & A \to C : (\mathsf{string}).C \to A : (\mathsf{formatC}).\mathsf{end} \end{array}$$

where the formatB is composed by one alphabet plus 6 natural numbers, while formatC is composed by 6 natural numbers. Protocols Buy and Book specify the sequence of interactions and the individual formats for transmitted data. A monitor can clearly realise that case (III) or (IV) coming from which principals and whether their contents are in right format.

6.1.2 Verifying Non-deterministic Results

Historically, most well-known process calculi, like CCS (115), CSP (79, 80), and the π-calculus (117, 118) assume the interactions happening among processes are synchronous. In comparison, the assumption that interactions are asynchronous makes analysis more challenging because (I) communications are hard to harness and (II) asynchronous interactions result in non-determinism. The following example shows the permutation mechanism, defined in Section 6.6, is useful for a monitor to verify non-deterministic results.

Example 6.1.1. Assume there are three processes in the network (introduced in Section 5.4) in parallel, each of which is separated by notation $\|$:

$$s[p,q]!\langle 1\rangle \parallel s[q,r]!\langle 2\rangle; s[p,q]?(x) \parallel s[q,r]?(x)$$

Let $P_1 = s[p,q]!\langle 1\rangle$, $P_2 = s[q,r]!\langle 2\rangle; s[p,q]?(x)$, and $P_3 = s[q,r]?(x)$. As the interactions are synchronous, the result of variable x is deterministic, which is 1. P_1 with action $s[p,q]!\langle 1\rangle$ can not take place until action $s[q,r]!\langle 2\rangle$ in P_2 takes place because its corresponding input $s[p,q]?(x)$ should wait until $s[q,r]!\langle 2\rangle$ fires, which is able to fire immediately because P_3 with action $s[q,r]?(x)$ is ready for it. Therefore, the sequence of actions is

$$s[q,r]!\langle 2\rangle \cdot s[q,r]?(x) \cdot s[p,q]!\langle 1\rangle \cdot s[p,q]?(x)$$

which is deterministic. When the interactions are *asynchronous*, P_1 and P_2 can take actions in any order or even concurrently. The sequence of actions can be

$$s[q,r]!\langle 2\rangle \cdot s[q,r]?(x) \cdot s[p,q]!\langle 1\rangle \cdot s[p,q]?(x) \quad \text{or} \quad s[p,q]!\langle 1\rangle \cdot s[p,q]?(x) \cdot s[q,r]!\langle 2\rangle \cdot s[q,r]?(x)$$

Thus the result of x may be 1 or 2 depending on P_2 with action $s[p,q]?(x)$ firstly receive the message from P_1 or P_3 with action $s[q,r]?(x)$ firstly receive the message from P_2.

Synchronous interactions, however, make the overall systems inefficient and do not reflect the facts of large-scale distributed systems.

As we focus on asynchronous interactions, to make an external monitor relying on specifications sensible to judge local processes' runtime behaviours, the permutation mechanism is proposed. The permutation mechanism can verify that both traces

$$s[q,r]!\langle 2\rangle \cdot s[q,r]?(x) \cdot s[p,q]!\langle 1\rangle \cdot s[p,q]?(x) \quad \text{or} \quad s[p,q]!\langle 1\rangle \cdot s[p,q]?(x) \cdot s[q,r]!\langle 2\rangle \cdot s[q,r]?(x)$$

are correct behaviours under asynchronous environment.

6.2 Global Specifications

Before introducing global specification with assertions, firstly the logical language for predicates (i.e. assertions) is defined below:

Definition 6.2.1 (logical language). The grammar for logical formulae (i.e. predicate) is:

$$A \ ::= \ \mathsf{true} \mid \mathsf{false} \mid e_1 = e_2 \mid e_1 > e_2 \mid \phi(e_1, ..., e_n) \mid A_1 \cap A_2 \mid \neg A \mid \exists x(A)$$

where $e_1, ...$ are expressions, and $\phi, ...$ range over a pre-defined set of predicates with fixed arities and types. $\exists x(A)$ returns true if there exists x in A; otherwise it returns false. Write $var(A)$ for the set of free variables occuring in A, similarly for $var(e)$.

Convention 6.2.2. Assume a fixed model for logical formulae satisfying: (I) validity of each closed atomic formula including equality and inequality is polynomially decidable; and (II) validity of closed formulae is decidable.

The syntax of global types, or called *global specifications*, is defined in Figure 6.1. It is based on (22), in which rule "interaction" is divided into the rules of "value" and "branching". In other words, the syntax in Figure 6.1 is not original but with some minor revision of the syntax of (22).

In Figure 6.1, sessions are described in terms of global specifications, which are called *types* in (86), and are called *assertions* in (22). Because, later in Chapter 8, we extend global/local assertions to embed *endpoint states* for specifying endpoints' behaviours through observing traces, for generality, from now on, a session type or a session assertion is uniformly called a *session specification* (i.e. global/local specifications).

$$
\begin{array}{rcll}
S & ::= & \mathsf{nat} \mid \mathsf{bool} \mid \mathtt{mode}(G[p]) & \text{sorts} \\
\mathtt{mode} & ::= & \mathtt{I} \mid \mathtt{O} \mid \mathtt{IO} & \text{modes} \\
G & ::= & p \to q : \{l_i(x_i : S_i)\langle\!\langle A_i \rangle\!\rangle.G_i\}_{i \in I} & \text{interaction} \\
 & \mid & \mu \mathsf{t}\langle \tilde{e} \rangle (\tilde{x} : \tilde{S})\langle\!\langle A \rangle\!\rangle.G \mid \mathsf{t}\langle e \rangle \mid \mathsf{end} & \text{rec/var/end} \\
 & \mid & G_1 \mid G_2 & \text{parallel}
\end{array}
$$

Figure 6.1: Global Specifications: G

Definition 6.2.3 (the set of roles in G). Define $role(G)$ as the set of roles occuring in global specification G. Whenever $p, q \in role(G)$, $p \neq q$.

The *global specification* G describes a session as a protocol signature among multiple roles with logical formulae. It is an abstract global scenario description which structures multi-party interactions and indicates the steps taken by the roles in the session. The main construct is a labelled message exchange, where p sends q a label l_i (I is a finite set of integers) and a message x_i with sort S_i. Sorts include base types and shared name type $\mathtt{mode}(G[p])$. In $\mathtt{mode}(G[p])$, G defines the behaviour of role p. G_i describes the continuation of the session for the selected branch i, and A_i is a predicate on interaction variables specifying what p must guarantee and dually what q can rely on. A_i expresses a constraint on the choice of branch i (e.g., a guard that must hold when selecting label l_i) and on the value of the exchanged message x_i, which we call *interaction variable*. We do not fix a specific logic for A, we only assume it is decidable. Interactions bind each x_i in A_i and in G_i.

A recursive specification is guarded by an invariant A, and defines a recursion parameter with its initialisation. $\tilde{e} = e_1, ..., e_n$, $\tilde{x} = x_1, ..., x_n$ and $\tilde{S} = S_1, ..., S_n$. and every x_i, x_j, $i \neq j$ are pairwise disjoint. $\tilde{x}$ is the binder of predicate A. Note that, the variable of recursion parameter is known by every participant involved in the recursion. For example, in recursion

$$\mu \mathsf{t}\langle e \rangle (x : S)\langle\!\langle A \rangle\!\rangle.p \to q : (y : \mathsf{int})\langle\!\langle y > x \rangle\!\rangle.\mathsf{t},$$

x is not an interaction variable sent from p to q but a parameter of this recursion, both p and q know x because they are all involved in the recursion. t is a variable for specifications. Its formal form is $\mathsf{t}\langle e \rangle$ because it may be a specification for a recursion

which has parameters. Branching and recursion describe potential choices and repeated interactions. Each message exchange/branching is annotated with a predicate specifying a constraint on the message value or the choice of a branch. end ends the session. Parallel composition is written as $G_1 \mid G_2$ in rule parallel. When a session follows $G_1 \mid G_2$, there are multi-threads running concurrently in this session. Note that whenever $G_1 \mid G_2$, $role(G_1) \cap role(G_2) = \emptyset$ holds. This structure provides the language flexibility, and benefits programers to write tasks in a readable structure.

Example 6.2.4 (OOI Instrument Commands - global specification). The following specification is the global specification for the example introduced in Chapter 5 (see Figure 5.2). A branch without predicate means its accompanying predicate is true. We sometimes omit labels when there is a single branch.

$$
\begin{array}{lcl}
G_{IC} & = & User \to Register : (x_{int} : \mathsf{InterfaceId}). \\
 & & Register \to User : (x_n : \mathsf{Int})\langle\!\langle x_n > 0 \rangle\!\rangle. User \to Agent : (x_p : \mathsf{Priority}). \\
 & & Agent \to User : \{\mathtt{accept}().G_{acc},\ \ \mathtt{reject}(x_E : \mathsf{ErrData})\langle\!\langle x_P = \mathsf{high} \supset x_e \neq \mathsf{busy} \rangle\!\rangle.\mathsf{end}\} \\
G_{acc} & = & \mu \mathsf{t}\langle x_n \rangle(y)\langle\!\langle y \geq 0 \rangle\!\rangle. \\
 & & User \to Agent : \{\mathtt{more}(x_{com} : \mathsf{Command})\langle\!\langle y > 0 \wedge x_{com} \neq \mathsf{switch\text{-}off} \rangle\!\rangle.G_{com}, \\
 & & \qquad\qquad \mathtt{quit}().Agent \to Instrument : \mathtt{quit}().\mathsf{end}\} \\
G_{com} & = & Agent \to Instrument : (y_{com} : \mathsf{Command}). \\
 & & Instrument \to Agent : (y_r : \mathsf{Response}).Agent \to User : (x_r : \mathsf{Response}).\ \mathsf{t}\langle y - 1 \rangle
\end{array}
$$

The scenario modelled by G_{IC} has already been described informally in Section 5.2, Chapter 5.

6.3 Consistency Principles

Based on (22, 86), this thesis addresses the consistency principles, well-formedness and well-assertedness for global specifications. For detailed theories and proofs, please see (22, 86).

The consistency principles of global specifications for ensuring the consistency of overall network behaviours are introduced in (22). This thesis, for completion reason, also mentions these principles which are defined on the basis of (22). The principles addressed are *history sensitivity and temporal satisfiability*. The well-formedness and well-assertedness of global specifications which are also very similar to those in (22), except the minor changes of the syntax of global specifications, are reviewed. Note, when we go to *stateful specifications*, Chapter 8, the revised stateful well-formedness for global

specifications will be introduced, while the well-assertedness for global specifications is still the same. Also note that the principle of locality, which is discussed in (22), is not addressed in this thesis because this thesis combines actions "interaction" and "branching" to be one action; which means, every action of "interaction" is one kind of action of "branching". Moreover, the principle of temporal satisfiability directly implies locality according to (22)'s Proposition B.10.

History sensitivity says, for every interaction,

1. the sender must know the variables/names he or she is going to send, and knows the variables/names deciding the result of the interaction predicate;
2. the receiver must know the variables/names, except those variables carried through the interaction (which are new to the receiver), deciding the result of the interaction predicate.

In other words, the interaction predicate of a participant must be defined only on those interaction variables introduced in the preceding interactions in which the participant is involved. Note that, according to the syntax of global specification shown in Figure 6.1, an interaction predicate is for both the sender and the receiver involved in this interaction. The variables carried through this interaction, which should be known by the sender, may be new to the receiver. The receiver knows these variables when he receives them.

Example 6.3.1 (history sensitivity). The following global specification violates history sensitivity

$$\begin{aligned} p \to q &: (x : \mathsf{int})\langle\!\langle x > 5 \rangle\!\rangle. \\ r \to q &: (y : \mathsf{int})\langle\!\langle y > x \rangle\!\rangle \end{aligned}$$

because r does not know variable x, which is only shared by roles p and q.

It can be easily revised to the global specification below

$$\begin{aligned} p \to q &: (x : \mathsf{int})\langle\!\langle x > 10 \rangle\!\rangle. \\ q \to r &: (x : \mathsf{int})\langle\!\langle \mathsf{true} \rangle\!\rangle. \\ r \to q &: (y : \mathsf{int})\langle\!\langle y > x \rangle\!\rangle \end{aligned}$$

to respect history sensitivity.

Well-formedness conditions for global specifications introduced later (Section 6.3.1) ensures a global specification satisfies history sensitivity.

As for temporal satisfiability, this principle says

> Every participant involved in an interaction should always find a valid path to proceed (in that interaction) unless he or she meets end.

Note, when interaction predicates do not attach to global specifications, like the ones in (86), a global specification always satisfies temporal satisfiability. When predicates are attached to interactions, they restrict the paths for interaction participants to go forward: Only the valid paths, according to the predicate, can forward the process.

The following example illustrates how a global specification violates temporal satisfiability principle, and how it can be revised to satisfy this principle.

Example 6.3.2 (temporal satisfiability). The global specification below violates temporal satisfiability

$$\begin{array}{lcl} p \to q & : & \mathrm{len}(x : \mathsf{int})\langle\!\langle x > 5 \rangle\!\rangle. \\ q \to p & : & \{\mathrm{walk}(y : \mathsf{int})\langle\!\langle x < y < 10 \rangle\!\rangle, \\ & & \quad \mathrm{run}(y : \mathsf{int})\langle\!\langle x < y < 50 \rangle\!\rangle\} \end{array}$$

because, when x is bigger than 50, which is actually possible since the predicate at role p only ensures $x > 5$, neither branch "walk" nor "run" can be selected for sending a proper y from role q to role p to satisfy the predicates specified at role q and p.

A revised one that respects temporal satisfiability can be

$$\begin{array}{lcl} p \to q & : & \mathrm{len}(x : \mathsf{int})\langle\!\langle 49 > x > 5 \rangle\!\rangle. \\ q \to p & : & \{\mathrm{walk}(y : \mathsf{int})\langle\!\langle x < y < 10 \rangle\!\rangle, \\ & & \quad \mathrm{run}(y : \mathsf{int})\langle\!\langle x < y < 50 \rangle\!\rangle\} \end{array}$$

where $x : \mathsf{int}$ and $49 > x > 5$ ensures the possible values of x are from 6 to 48. Even when a value in the range of 8 to 48 is sent from role p to role q, q can select branch "run" to send y in the range of 9 to 49 to satify the predicate $x < y < 50$.

When the local endpoint states are embeded into specifications, the consistency principles are revised, which are discussed in Section 8.2.2, Chapter 8.

6.3.1 Well-formedness

The well-formedness of global specifications stipulates the principle of *history sensitivity*, which can be ensured linearly through a syntactic checker (for details, please see (22)). Before introducing well-formedness, a convention and definitions are stated as follows.

Convention 6.3.3. This thesis assumes the standard bound name convention for all syntactic entities with bindings.

$\mathcal{I}(G)$ is defined as the one in (22):

Definition 6.3.4. Define $\mathcal{I}(G)$ as the set of interaction variables occurring in G. $p \in role(G)$ knows an interaction variable $u \in \mathcal{I}(G)$ when p sends u or receives u, or when u is a parameter of a recursion where p involves.

Definition 6.3.5 (projection of interaction variables, $\mathcal{I}(G) \upharpoonright p$). Define $\mathcal{I}(G) \upharpoonright p$, where $p \in role(G)$, as the set of variables that role p knows.

Definition 6.3.6. Let $\tilde{S} = S_1, ..., S_i, ..., S_n$. Define $\mathtt{t} : \tilde{S}$ for representing that the sort of ith parameter of $\mathtt{t}$ is S_i.

Definition 6.3.7 (shared environments). Let Γ be the shared environments, defined by the following grammar:

$$\Gamma ::= \emptyset \mid \Gamma, x : S \mid \Gamma, u : \mathtt{mode}(G[p]) \mid \Gamma, \mathtt{t} : \tilde{S}$$

where x is a variable for integer, string, and boolean, etc., while u is a variable for shared name, and $\mathtt{t}$ is for specification variable, When it is used to handle a recursive specification, it has the corresponding parameters to the recursion specification. Note that, $\Gamma \vdash e$ if for every $x \in var(e)$, $\Gamma \vdash x$.

As the one defined in (22), the well-formedness for global specifications is defined in Figure 6.2 with some revisions to cater for the syntax of our global specifications.

The key rule [WF-interaction] requires that interaction participants must know all the variables appearing in predicate A.

$$\forall x.((x \in \cup_{i \in I} var(A_i) \ \cap \ x \notin \{x_i\}_{i \in I}) \supset x \in (\mathcal{I}(G) \upharpoonright p \ \cap \mathcal{I}(G) \upharpoonright q))$$

means that for any x which is in the assertions $\{A_i\}_{i \in I}$ but not as an interaction variable, it should be known by both roles p and q. In rule [WF-rec], $\mathtt{t} : S_1, ..., S_n$

$$\frac{\begin{array}{c}\forall i \in I, \Gamma, x_i : S_i \vdash G_i \\ \forall x.((x \in \cup_{i\in I} var(A_i) \ \cap \ x \notin \{x_i\}_{i\in I}) \supset x \in (\mathcal{J}(G) \upharpoonright p \ \cap \mathcal{J}(G) \upharpoonright q))\end{array}}{\Gamma \vdash p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G_i\}_{i\in I}} \quad \text{[WF-interaction]}$$

$$\frac{\Gamma \vdash G \quad \Gamma \vdash G'}{\Gamma \vdash G \mid G'} \quad \text{[WF-par]}$$

$$\frac{}{\Gamma \vdash \mathsf{end}} \quad \text{[WF-end]}$$

$$\frac{\Gamma \vdash e_1 : S_1, ..., \Gamma \vdash e_n : S_n}{\Gamma, \mathsf{t} : S_1, ..., S_n \vdash \mathsf{t}\langle e_1, ..., e_n\rangle} \quad \text{[WF-rec]}$$

$$\frac{\Gamma' = \Gamma, \mathsf{t} : S_1, ..., S_n \quad \Gamma' \vdash G \quad \mathsf{dom}(\Gamma') \supseteq var(A) \quad \forall i.\Gamma \vdash x_i : S_i, e_i : S_i}{\Gamma \vdash \mu\mathsf{t}\langle e_1, ..., e_n\rangle(x_1 : S_1, ..., x_n : S_n)\langle\langle A\rangle\rangle.G} \quad \text{[WF-def]}$$

Figure 6.2: Well-formedness for Global Specifications

means the parameters of recursive variable t has types $S_1, ..., S_n$. This knowledge is known by the participants involved in the recursion. [WF-def] says that, if the shared environment can prove the value of each parameter follows the specified type, i.e. $\forall i.\Gamma \vdash x_i : S_i, e_i : S_i$, and prove the recursion body G, i.e. $\Gamma' \vdash G$ where $\Gamma' = \Gamma, \mathsf{t} : S_1, ..., S_n$, which implies that the invariant A is true, then it can prove the recursive specification $\mu\mathsf{t}\langle e_1, ..., e_n\rangle(x_1 : S_1, ..., x_n : S_n)\langle\langle A\rangle\rangle.G$. Other rules are straightforward. According to (22), the rules are purely syntactic, and the validation of G is a linear time problem.

6.3.2 Well-assertedness

Well-assertedness is defined in terms of principle of temporal satisfiability. Temporal satisfiability requires that for every possible value sent and branch selected in the past, it is always possible to make a choice that satisfies the current predicate. The most trivial violation of temporal satisfiability is an interaction with one single branch and predicate false. The intuition for realising temporal satisfiability is to ensure that, for every interaction predicate A, there exists an interaction predicate A' just appearing after A such that A implies A', i.e. $A \supset A'$. Below the checker for temporal satisfiability, a boolean function $GSat$, is defined based on (22).

Definition 6.3.8 (well-assertedness for global specifications). Assume at the beginning $A = \mathsf{true}$. We define a recursively boolean function $GSat(G, A)$ as follows:

(i) If $G = p_1 \to p_2 : \{l_i(x_i : S_i)\langle\!\langle A_i \rangle\!\rangle.G_i\}_{i \in I},\ I = \{1, ..., n\}$ then $GSat(G, A)$ returns

$$\begin{cases} GSat(G_1, A \cap A_1) \cap \ ... \cap GSat(G_n, A \cap A_n) & \text{if } A \supset \bigcup_{i \in I}(\exists x_i(A_i)) \\ \mathsf{false} & \text{otherwise} \end{cases}$$

(ii) $G = G_1 \mid G_2$ then $GSat(G, A)$ returns $GSat(G_1, A) \cap GSat(G_2, A)$.

(iii) $G = \mu \mathsf{t}\langle \tilde{e} \rangle (\tilde{x} : \tilde{S})\langle\!\langle A' \rangle\!\rangle.G'$ then $GSat(G, A)$ returns

$$\begin{cases} GSat(G', A \cap A') & \text{if } A \supset A'\{\tilde{e}/\tilde{x}\} \\ \mathsf{false} & \text{otherwise} \end{cases}$$

(iv) $G = \mathsf{t}_{(\tilde{x}, A')}\langle \tilde{e} \rangle$ then $GSat(G, A)$ returns true if $A \supset A'\{\tilde{e}/\tilde{x}\}$ and false otherwise.

(v) $G = \mathsf{end}$ then $GSat(G, A)$ returns true.

Write an annotation of type variable t as $\mathsf{t}_{(\tilde{x}, A')}\langle \tilde{e} \rangle$ to represent that type variable t has a predicate A' (i.e. recursion invariant) associated to this recursion and parameters $\tilde{x}$. This annotation is always possible if t is bound in a whole global specification, i.e. if the whole global specification is closed.

In (i), it requires that there exists at least one branch, say label j, that can be chosen such that $\exists x_j(A_j)$ is true whenever A is true. For example,

$$p \to q \quad : \quad \{(x : \mathsf{int})\langle\!\langle x < 10 \rangle\!\rangle.G'_1, (x : \mathsf{int})\langle\!\langle x \geq 10 \rangle\!\rangle.G'_2\}$$

is well-asserted because if the first branch fails, the second branch can be chosen, and vice versa. Also,

$$p \to q \quad : \quad \{(x : \mathsf{int})\langle\!\langle x > 10 \rangle\!\rangle.G'_1, (y : \mathsf{int})\langle\!\langle y > 10 \rangle\!\rangle.G'_2\}$$

is well-asserted when it starts from $A = \mathsf{true}$ because p is able to send valid values for variables x and y. In (ii), a parallel composition is satisfiable if both of them are satisfiable. (iii) and (iv) require A to imply the predicate A' of the recursion after replacing variables $\tilde{x}$ with $\tilde{e}$ in the recursion body (i.e. initiation parameters). (v) says a session termination end always returns true.

$$
\begin{array}{rcll}
T & ::= & p!\{l_i(x_i{:}S_i)\langle\!\langle A_i\rangle\!\rangle.T_i\}_{i\in I} & \text{selection} \\
 & | & p?\{l_i(x_i{:}S_i)\langle\!\langle A_i\rangle\!\rangle.T_i\}_{i\in I} & \text{branch} \\
 & | & \mu \mathsf{t}\langle e\rangle(x:S)\langle\!\langle A\rangle\!\rangle.T \;\mid\; \mathsf{t}\langle e\rangle \;\mid\; \mathsf{end} & \text{rec/var/end}
\end{array}
$$

Figure 6.3: Local Specifications: T

6.4 Local Specifications

For runtime monitoring, every local process is associated a dedicated monitor that manages its interactions with the network (hence with other peers) by inspecting outgoing and incoming messages.

The *endpoint specifications*, T, are defined in Figure 6.3. They are local specification used by monitors to represent which interactions are acceptable for an endpoint. A local specification describes a session from the perspective of a role.

Rule selection expresses the transmission to role p of a label l_i taken from a label pool $\{l_i\}_{i\in I}$, where I is a set of indexes, together with an interaction variable x_i of sort S_i and that the remaining interaction in the session is T_i. Branching is its dual. Others are similar to the global version.

Based on Example 6.2.4, Example 6.4.1 shows the endpoint specifications (which can be automatically projected from G_{IC}. The projection algorithm is introduced in Section 6.5.).

Example 6.4.1 (OOI Instrument Commands - the user endpoint).

$$
\begin{array}{rcl}
T_{user} & = & \mathit{Register}!(x_{int} : \mathsf{InterfaceId}).\mathit{Register}?(x_n : \mathsf{Int})\langle\!\langle x_n > 0\rangle\!\rangle.\mathit{Agent}!(x_p : \mathsf{Priority}). \\
 & & \mathit{Agent}?\{\mathtt{accept}().T^u_{acc},\ \ \mathtt{reject}(x_e : \mathsf{ErrData})\langle\!\langle x_p = \mathsf{high} \supset x_e \neq \mathsf{busy}\rangle\!\rangle.\mathsf{end}\} \\
T^u_{acc} & = & \mu \mathsf{t}\langle x_n\rangle(y)\langle\!\langle y \geq 0\rangle\!\rangle. \\
 & & \mathit{Agent}!\{\mathtt{more}(x_{com} : \mathsf{Command})\langle\!\langle y > 0 \wedge x_{com} \neq \mathsf{switch\text{-}off}\rangle\!\rangle.T^u_{com},\ \ \mathtt{quit}().\mathsf{end}\} \\
T^u_{com} & = & \mathit{Agent}?(x_r : \mathsf{Response}).\mathsf{t}\langle y - 1\rangle
\end{array}
$$

Well-assertedness for endpoint specifications is defined in the same way as the one for global specifications:

Definition 6.4.2 (well-assertedness for local specifications). Assume at the beginning $A = \mathsf{true}$. We define a recursively boolean function $GSat(T, A)$ as follows:

(i) If $T = p!\{l_i(x_i : S_i)\langle\!\langle A_i\rangle\!\rangle.T_i\}_{i\in I}$ or $T = p?\{l_i(x_i : S_i)\langle\!\langle A_i\rangle\!\rangle.T_i\}_{i\in I}$ where $I =$

$\{1, ..., n\}$, then $GSat(T, A)$ returns

$$\begin{cases} GSat(T_1, A \cap A_1) \cap \ ... \cap GSat(T_n, A \cap A_n) & \text{if } A \supset \bigcup_{i \in I}(\exists x_i(A_i)) \\ \mathsf{false} & \text{otherwise} \end{cases}$$

(ii) $T = \mu \mathsf{t}\langle \tilde{e} \rangle (\tilde{x} : \tilde{S}) \langle\!\langle A' \rangle\!\rangle . T'$ then $GSat(T, A)$ returns

$$\begin{cases} GSat(T', A \cap A') & \text{if } A \supset A'\{\tilde{e}/\tilde{x}\} \\ \mathsf{false} & \text{otherwise} \end{cases}$$

(iii) $T = \mathsf{t}_{(\tilde{x}, A')}\langle \tilde{e} \rangle$ then $GSat(T, A)$ returns true if $A \supset A'\{\tilde{e}/\tilde{x}\}$ and false otherwise.

(iv) $T = \mathsf{end}$ then $GSat(T, A)$ returns true.

In (i), it is required that, either for sending or receiving, there exists at least one branch, say label j, that can be chosen such that $\exists x_j(A_j)$ is true whenever A is true. Other definitions for well-assertedness for local specifications are for same reasons as the ones defined for the case of global specifications.

Together with the calculus defined in Chapter 5, a monitor with T can specify the behaviours of local processes. Once a process joins a session, the corresponding monitor is activated and enforces conformance to the scenario of the subsequent interactions prescribed for the process to follow.

6.5 Projection from the Global to the Local

In this section we define the projection from a global specification to endpoint specifications. Projectability is a structural property from the underlying type (86) and well-assertedness is a property on predicates ensuring it is always possible for an endpoint to satisfy its own obligations (22). This thesis assumes these properties for specifications. Figure 6.4 illustrates the relationships between global specifications and local specifications linked by projection. A local specifications is projected to a role specified in a global specification. It is either used for runtime monitoring or for static type checking. An *endpoint projection* or often simply *projection*, denoted by $G \upharpoonright p$, projects G onto p returning an endpoint specification. The projection in this thesis is defined similarly

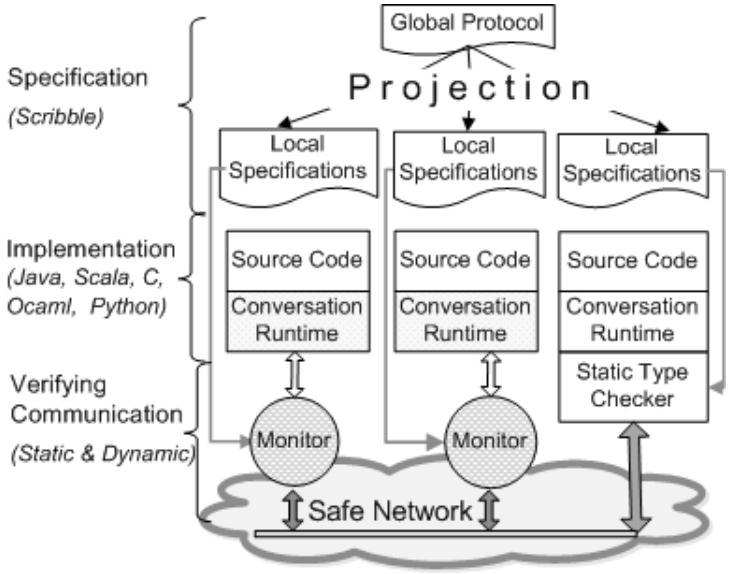

Figure 6.4: The Relationships between the Global Specifications, Local Specificaitons, and the Projection

to the one proposed in (22). Some rules are revised according to the combination of actions of "interaction" and "branching".

With Definition 6.3.4, $\mathcal{J}(G)$ is the set of interaction variables exchanged and recursion parameters in G. We let $\mathcal{J}(G) \upharpoonright p$ be the set of variables known by p which are either: (I) the variables that p sends or receives in G, or (II) the parameter of recursions e.g., $\mu \mathtt{t}\langle e \rangle (y) \langle\!\langle A \rangle\!\rangle . P$, where p knows all the variables in the expression e and p knows all variables in e' for all $\mathtt{t}\langle e' \rangle$ in P. We define the projection of a predicate on a participant as $A \upharpoonright p = \exists V_{ext} . A$ with $V_{ext} = var(A)/(\mathcal{J}(G) \upharpoonright p)$ (i.e., closing with the existential quantifier the free variables of A that p does not know).

Based on (22), the projection of a global specification onto each role with an end-point specification is defined as a recursive function $Proj$. It is also extended to the mergeability studied in (50). We use annotation $\mathtt{t}_x$ for recursive call where x is the recursion parameter for $\mathtt{t}$ (i.e., in $\mu \mathtt{t}\langle e \rangle (x : S) \langle\!\langle A \rangle\!\rangle . G$).

Definition 6.5.1 (Projection). Assume $p \in G$, $p_1 \neq p_2$. Let $A \downarrow \mathsf{true}$. The projection of G on p, written $Proj(G, A, p) = G \upharpoonright p$, is defined as follows.

1. $Proj(p_1 \to p_2 : \{l_i(x_i : S_i)\langle\!\langle A_i \rangle\!\rangle.G_i\}, A_{proj}, p) =$

$$\begin{cases} p_2!\{l_i(x_i : S_i)\langle\!\langle A_i \rangle\!\rangle.G^i_{proj}\}_{i\in I} & \text{if } p = p_1 \neq p_2, \\ p_1?\{l_i(x_i : S_i)\langle\!\langle (A_i \cap A_{proj}) \upharpoonright p \rangle\!\rangle.G^i_{proj}\}_{i\in I} & \text{if } p = p_2 \neq p_1, \\ \sqcup_{i\in I} G^i_{proj} & \text{if } p \neq p_1, p \neq p_2 \end{cases}$$

with $G^i_{proj} = Proj(G_i, A_i \cap A_{proj}, p)$.

2. $Proj((G_1 \mid G_2), A_{proj}, p) =$

$$\begin{cases} Proj(G_i, A_{proj}, p) & \text{if } p \in G_i \text{ and } p \notin G_j, i \neq j \in \{1, 2\} \\ \mathsf{end} & \text{if } p \notin G_1 \text{ and } p \notin G_2. \end{cases}$$

3. $Proj(\mu \mathsf{t}\langle v \rangle(x\!:\!S)\langle\!\langle A \rangle\!\rangle.G, A_{proj}, p) =$

$$\begin{cases} \mu \mathsf{t}\langle v \rangle(x\!:\!S)\langle\!\langle A \upharpoonright p \rangle\!\rangle.Proj(G, A \cap A_{proj}, p) & \text{if } x \in \mathcal{I}(G) \upharpoonright p \\ \mu \mathsf{t}\langle\!\langle A \upharpoonright p \rangle\!\rangle.Proj(G, A \cap A_{proj}, p) & \text{if } x \notin \mathcal{I}(G) \upharpoonright p \text{ and } p \in role(G) \\ \mathsf{end} & \text{if } p \notin role(G) \end{cases}$$

4. $Proj(\mathsf{t}_x\langle v \rangle, A_{proj}, p) =$

$$\begin{cases} \mathsf{t}_x\langle v \rangle & \text{if } x \in \mathcal{I}(G) \upharpoonright p \\ \mathsf{t}_x & \text{otherwise} \end{cases}$$

5. $Proj(\mathsf{end}, A_{proj}, p) = \mathsf{end}$.

where $\sqcup_{i\in I} G^i_{proj} = G^1_{proj} \sqcup G^2_{proj} ... \sqcup G^n_{proj}$ if $I = \{1, ..., n\}$. The mergeability, $T_1 \sqcup T_2$, is defined by $T \sqcup T = T$ and the following axiom based on (50). Let $\dagger \in \{!, ?\}$.

$$\begin{array}{l} q \dagger \{l_i(x_i\!:\!S_i)\langle\!\langle A_i \rangle\!\rangle.T_i\}_{i\in I} \ \sqcup \ q \dagger \{l'_j(x'_j\!:\!S'_j)\langle\!\langle A'_j \rangle\!\rangle.T'_j\}_{j\in J} = \\ q \dagger \{l_k(x_k\!:\!S_k)\langle\!\langle A_k \rangle\!\rangle.T_k\}_{k\in I\setminus J} \ \cup \ q \dagger \{l'_k(x'_k\!:\!S'_k)\langle\!\langle A'_k \rangle\!\rangle.T'_k\}_{k\in J\setminus I} \cup \\ q \dagger \{l_k(x_k\!:\!S_k)\langle\!\langle A_k \cap A'_k \rangle\!\rangle.T_k \sqcup T'_k\}_{k\in I\cap J} \end{array}$$

when $\forall k \in I \cap J, S_k = S'_k$. Then we close the local specification context, defined in Definition 6.6.7, $\mathfrak{T}[T_1] \sqcup \mathfrak{T}[T_1] = \mathfrak{T}[T_1 \sqcup T_2]$.

In rule 1 for an output the projection of a predicate A consists of A itself. Note that if G is well-formed then p_1 knows all variables in A. For a receiver, it is not sufficient to verify the non-violation of the current predicate only. Consider the following global

specification:

$$Seller \to Buyer : (cost : \mathsf{Int})\langle\!\langle cost > 10 \rangle\!\rangle.Buyer \to Bank : (pay : \mathsf{Int})\langle\!\langle pay \geq cost \rangle\!\rangle$$

The predicate $pay \geq cost$ is meaningless to $Bank$ since $Bank$ does not know $cost$; rather the projection on $Bank$ should be $Buyer?(pay : \mathsf{Int})\langle\!\langle \exists cost(cost > 10 \wedge pay \geq cost) \rangle\!\rangle$, which incorporates the constraint between $Buyer$ and $Seller$. Thus rule 1 projects all the past predicates while hiding incorporating the constraints on interactions p_2 does not participate through existential quantification.

In rule 3, the case when $p \notin role(G)$, then it returns end for ending this session. For example, as we project $\mu \mathsf{t}.p \to q : (x : \mathsf{int}).\mathsf{t}$ to role r, and $r \neq p$ and $r \neq q$, the recursion should terminate by denoting with end. Other rules are straightforward.

According to (22), the projections of a well-formed global assertion are also well-asserted. In Example 6.2.4, global specification G_{IC} is well-formed and well-asserted, then it is projectable and the endpoint specifications projected from G_{IC} are all well-asserted. The lemma proposed and proved in (22) is mentioned below:

Lemma 6.5.2. *If G is well-formed then, for all predicates A_G, A_T such that $A_T \supset A_G$ and all $p \in role(G)$,*

$$GSat(G, A_G) \supset GSat(G \upharpoonright p, A_T)$$

6.6 Permutation of Specifications

The permutation mechanism is introduced for runtime monitoring, which needs to verify a non-deterministic sequence of actions due to the asynchronous interactions among processes and monitors.

In this section, we formalise the notion of minimum permutations for monitors. The key idea is to "permute up" an action which is in effect *not* suppressed by another action to the top, without affecting other actions. Such an action which eventually moves to the top can originally occur in many places in an assertion: These occurrences are merged into one when a sequence of permutations are applied. Here firstly introduces several preliminary notions.

Definition 6.6.1 (global specification context). A *global specification context* $\mathcal{G}[\,]$

is a global specification with a hole, whose grammar is defined as:

$$\begin{array}{rcl} \mathcal{G}[\,] & ::= & [\,] \mid \mathcal{G}[\,]|G' \mid G'|\mathcal{G}[\,] \\ & \mid & p_1 \to p_2 : \{l_1(x_1 : S_1)\langle\!\langle A_1 \rangle\!\rangle.G_1, .., l_i(x_i : S_i)\langle\!\langle A_i \rangle\!\rangle.\mathcal{G}[\,], .., l_n(x_n : S_n)\langle\!\langle A_n \rangle\!\rangle.G_n\} \end{array}$$

Then $\mathcal{G}[G]$ denotes the result of filling the hole in the global specification context $\mathcal{G}$ with global specification G. Whenever write $\mathcal{G}[G]$, $\mathcal{G}[G]$ is assumed well-formed. Note that $[\,]$ defines the identity function, i.e. $[G] = G$.

Note that there is no need for having $\mu t.\mathcal{G}[\,]$ because such a recursion can always be unfolded into each component.

Definition 6.6.2. Define a closed global specification G is *prime* if $G \neq \mathsf{end}$ and $G = G_1|G_2$ implies one of $G_i, i = \{1, 2\}$ is end. If G is a prime specification, then the *initial action* of G is the action at the top of G. For a non-prime G, its *initial actions* are those of its prime components. The initial action of a closed local specification T is defined similarly.

Example 6.6.3. Let

$$G = p_1 \to q_1 : \{l_i(x_i : S_i)\langle\!\langle A_i \rangle\!\rangle.p_2 \to q_2 : \{l'_j(x'_j : S'_j)\langle\!\langle A'_j \rangle\!\rangle.G'_j\}_{j\in J}\}_{i\in I}.$$

The *initial action* of G is $p_1 \to q_1 : \{l_i(x_i : S_i)\langle\!\langle A_i \rangle\!\rangle\}_{i\in I}$. Another example is, let $G = G_1 \mid G_2$, $G_1 = p_1 \to q_1 : \{l_i(x_i : S_i)\langle\!\langle A_i \rangle\!\rangle.G_i\}_{i\in I}$, and $G_2 = p_2 \to q_2 : \{l'_j(x'_j : S'_j)\langle\!\langle A'_j \rangle\!\rangle.G'_j\}_{j\in I}$. The *initial actions* of G are $p_1 \to q_1 : \{l_i(x_i : S_i)\langle\!\langle A_i \rangle\!\rangle\}_{i\in I}$ and $p_2 \to q_2 : \{l'_j(x'_j : S'_j)\langle\!\langle A'_j \rangle\!\rangle\}_{j\in I}$.

The idea of *marking actions* in global/local specifications is introduced, so that the effect of each permutation can be tracked. Hereafter, the specification context also includes the marked actions.

Definition 6.6.4 (specifications with marked actions).

1. *A global specification with marked actions* or for brevity, *a marked global specification*, is a global specification some of whose action occurrences are underlined actions, denoted by underlining the initial part of the action. Write $(\underline{G}, \ldots)$ for marked global specifications. The underlined actions in $\underline{G}$ are called *permutation candidates of* $\underline{G}$, e.g. $\mathcal{G}[\underline{p \to q} : \{l_i(x_i : S_i)\langle\!\langle A_i \rangle\!\rangle.G_i\}_{i\in I}]$ is a marked global specification, and $\underline{p \to q}$ is its permutation candidate.

2. Similarly, define marked local specifications ($\underline{T}, \ldots$) as e.g.

$$\mathcal{T}[\underline{p}!\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.T_i\}_{i\in I}],$$

where $\underline{p}$ is its permutation candidate.

First define a unit permutation, which is a permutation of two nested branchings defined only when they are independent, i.e. there is no causal order between two interactions (no causality through either ordered actions from the same sender to the same receiver, or ordered actions linked by a receiver becoming a sender in the second action).

Definition 6.6.5 (global unit permutation). A *unit permutation* of marked global specifications is defined by the axiom below:

$$\begin{aligned}&\mathcal{G}[p_1 \to p_2 : \{l_i(x_i{:}S_i)\langle\langle A_i\rangle\rangle.\underline{p_3 \to p_4} : \{l'_j(x'_j{:}S'_j)\langle\langle A'_j\rangle\rangle.G_{ij}\}_{j\in J}\}_{i\in I}]\\ &\quad \curvearrowright^1 \mathcal{G}[\underline{p_3 \to p_4} : \{l'_j(x'_j{:}S'_j)\langle\langle A'_j\rangle\rangle.p_1 \to p_2 : \{l_i(x_i{:}S_i)\langle\langle A_i\rangle\rangle.G_{ij}\}_{i\in I}\}_{j\in J}]\end{aligned}$$

with the following conditions: if $(p_2 = p_4)$ then $(p_1 \neq p_3)$; if $(p_1 = p_4)$ then $(p_2 \neq p_3)$; if $(p_1 = p_3)$ then $(p_2 \neq p_4)$; or p_i pairwise distinct.

Remark 6.6.6 (unit permutation of global specifications).

1. Definition 6.6.5 demands each specification under permutation candidate, G_{ij}, to be unmarked specifications (note the symbol G_{ij} is for an unmarked specification).

2. In the same definition, the inner actions to be permuted up, should have precisely the same branching labels. This condition is automatically ensured by projectability.

The second point above is significant in that, whenever there is no causality among actions, they automatically enable a unit permutation: An inner, nested but causally unsuppressed action can automatically be permuted up over another action which happens to be above it.

Definition 6.6.7 (local specification context). A *local specification context* $\mathcal{T}[\ ]$ is a local specification with a hole, whose grammar is defined as:

$$\begin{aligned}\mathcal{T}[\ ] \quad ::= \quad & [\ ] \mid p?\{l_1(x_1{:}S_1)\langle\langle A_1\rangle\rangle.T_1, .., l_i(x_i{:}S_i)\langle\langle A_i\rangle\rangle.\mathcal{T}[\ ], .., l_n(x_n{:}S_n)\langle\langle A_n\rangle\rangle.T_n\}\\ \mid \quad & p!\{l_1(x_1{:}S_1)\langle\langle A_1\rangle\rangle.T_1, .., l_i(x_i{:}S_i)\langle\langle A_i\rangle\rangle.\mathcal{T}[\ ], .., l_n(x_n{:}S_n)\langle\langle A_n\rangle\rangle.T_n\}\end{aligned}$$

Then $\mathcal{T}[T]$ denotes the result of filling the hole in the local specification context $\mathcal{T}$ with global specification T. Note that $[\ ]$ defines the identity function, i.e. $[T] = T$.

In the same way as for global specifications, define the unit permutations for local specifications. In this case, the causal order only occurs when there are two consecutive sending actions with the same target, or when there is an input action followed by an output action. These conditions precisely correspond to permutability in global specifications through projections.

Definition 6.6.8 (local unit permutation). The *unit permutation* of local (endpoint) specifications by the axioms below, assuming $p_1 \neq p_2$:

$$\begin{aligned}
&\mathcal{T}[p_1!\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.\underline{p_2!}\{l'_j(x'_j : S'_j)\langle\langle A'_j\rangle\rangle.T_{ij}\}_{j\in I}\}_{i\in I}] \\
&\qquad\qquad \curvearrowright^1 \quad \mathcal{T}[\underline{p_2!}\{l'_j(x'_j : S'_j)\langle\langle A'_j\rangle\rangle.p_1!\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.T_{ij}\}_{i\in I}\}_{j\in I}] \\
&\mathcal{T}[p_1?\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.\underline{p_2?}\{l'_j(x'_j : S'_j)\langle\langle A'_j\rangle\rangle.T_{ij}\}_{j\in I}\}_{i\in I}] \\
&\qquad\qquad \curvearrowright^1 \quad \mathcal{T}[\underline{p_2?}\{l'_j(x'_j : S'_j)\langle\langle A'_j\rangle\rangle.p_1?\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.T_{ij}\}_{i\in I}\}_{j\in I}] \\
&\mathcal{T}[p_1!\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.\underline{p_2?}\{l'_j(x'_j : S'_j)\langle\langle A'_j\rangle\rangle.T_{ij}\}_{j\in I}\}_{i\in I}] \\
&\qquad\qquad \curvearrowright^1 \quad \mathcal{T}[\underline{p_2?}\{l'_j(x'_j : S'_j)\langle\langle A'_j\rangle\rangle.p_1!\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.T_{ij}\}_{i\in I}\}_{j\in I}]
\end{aligned}$$

Remark 6.6.9 (unit permutation of local specifications).

1. As before, demand each specification under permutation candidate to be unmarked.
2. The inner (lower) actions have precisely the same branching labels, automatically ensured among projected local specifications.

We introduce the minimum permutation relation in the follows. Below write $\mathsf{unmark}(\underline{G})$ for the result of taking off marks from the actions in $\underline{G}$, similarly for $\mathsf{unmark}(\underline{T})$.

Definition 6.6.10 (minimum permutation). Let $\underline{G}$ and $\underline{G'}$ be marked. Then define the *minimum permutation from* $\underline{G}$ *to* $\underline{G'}$, denoted $\underline{G} \curvearrowright \underline{G'}$, by the following induction.

$$\frac{\text{all and only marked action in } G \text{ is one of its initial actions.}}{\underline{G} \curvearrowright \underline{G}}$$

$$\frac{\underline{G} \curvearrowright^1 \underline{G''} \quad \underline{G''} \curvearrowright \underline{G'}}{\underline{G} \curvearrowright \underline{G'}}$$

Then define $G \curvearrowright G'$ by:

$$\frac{G = \mathsf{unmark}(\underline{G}) \quad \underline{G} \curvearrowright \underline{G}' \quad G' = \mathsf{unmark}(\underline{G}')}{G \curvearrowright G'}$$

Similarly define $\underline{T} \curvearrowright \underline{T}'$ by:

$$\frac{\text{all and only marked action in } \underline{T} \text{ is its initial action.}}{\underline{T} \curvearrowright \underline{T}}$$

$$\frac{\underline{T} \curvearrowright^1 \underline{T}'' \quad \underline{T}'' \curvearrowright \underline{T}'}{\underline{T} \curvearrowright \underline{T}'}$$

As before, define $T \curvearrowright T'$ by unmarking.

Basic properties of the minimum permutations follow. In brief, these results say that marking does not determine minimum permutations.

Proposition 6.6.11 (minimum permutation to the top).

1. The minimum permutation to permute a marked action in $\underline{G}$ to the top is inductively applying Definition 6.6.10 until the marked action is at the top of $\underline{G}$.
2. Whenever $\underline{G} \curvearrowright \underline{G}'$ and $\underline{G}' \curvearrowright \underline{G}'$, there is always a unique marked action in $\underline{G}'$, which is one of its initial actions. Similarly whenever $\underline{T} \curvearrowright \underline{T}'$, there is always a unique marked action in $\underline{T}'$, which is its initial action.
3. If $\underline{G} \curvearrowright \underline{G}'_1$, $\underline{G} \curvearrowright \underline{G}'_2$, $\underline{G}'_1 \curvearrowright \underline{G}'_1$ and $\underline{G}'_2 \curvearrowright \underline{G}'_2$, then $\underline{G}_1 = \underline{G}_2$. Similarly if $\underline{T} \curvearrowright \underline{T}'_1$ and $\underline{T} \curvearrowright \underline{T}'_2$, then $\underline{T}_1 = \underline{T}_2$.
4. If $G \curvearrowright G'$ and $\mathsf{unmark}(\underline{G}) = G$, then there is a unique $\underline{G}'$ such that $\underline{G} \curvearrowright \underline{G}'$ and $\mathsf{unmark}(\underline{G}') = G'$. Similarly, if $T \curvearrowright T'$ and $\mathsf{unmark}(\underline{T}) = T$, then there is a unique $\underline{T}'$ such that $\underline{T} \curvearrowright \underline{T}'$ and $\mathsf{unmark}(\underline{T}') = T'$.

By Proposition 6.6.11, write $G \curvearrowright \underline{p \to q} : \{l_i(x_i : S_i)\langle\langle A_i \rangle\rangle.G_i\}_{i \in I}]$ to indicate a unique permutation from G: Just identifying the resulting top action is enough to determine the minimum permutation.

7

The Calculus of Dynamic Asynchronous Monitoring

Overview Based on the calculus proposed in Chapters 5 and the specification language introduced in Chapter 6, this chapter introduces a realistic yet formal model of the monitored network in which strong safety properties result from local runtime verification and enforcement of possibly untrusted endpoints, rather than from static checking. The monitoring model proposed here is proven to guarantee the satisfaction of global specifications, focusing on the following properties: The dynamic local and global safety through stating local and global communication safety (Theorems 7.4.3 and 7.4.16); the external local and global monitoring transparency (Theorems 7.4.5 and 7.4.17); and session fidelity (Theorem 7.4.28) in the presence of possibly ill-behaved processes.

We start from Section 7.1, in which motivating examples and a general framework for monitoring are introduced. Section 7.2 is one main part of this chapter: It introduces the syntax and semantics of external monitors and provides viable and efficient reference semantics for monitor implementers. Then it introduces the syntax and semantics of monitor-off, which can be viewed as a gateway. It finally introduces the syntax and semantics of monitored network, which consists of local monitored processes and the global transport. Section 7.3 provides two use cases to illustrate how monitoring works. As another main part of this chapter, Section 7.4 states and proves (the full proofs are in Appendix B) the theorems of local and global safety, local and global transparency, and session fidelity. Section 7.5 introduces a global observer, denoted by $\mathfrak{E}$, which formalises the global observable interactions from the global point of view. We can also apply $\mathfrak{E}$

to ensure global safety and session-fidelity through observing the global transport and the collection of network monitors.

7.1 Motivating Example and the Framework of Monitoring

Many non-trivial safety properties of distributed applications can only be guaranteed when the endpoints are reciprocally assured of their cooperation (i.e. after static checking). For example, by the adherence to an application-level protocol ("message choreography" (35, 156)), they need to satisfy *global invariants* over the behaviour of multiple peers. Many *global invariants* can only be effectively assured through *runtime local verification* since a centralised verification is unfeasible in large-scale distributed systems, due to e.g. latency.

7.1.1 Motivating Example

In this section, two examples are introduced to show the effectiveness of runtime local verification. One example shows that, based on the specifications defined in Chapter 6, during runtime, a monitor can easily detect a possible malicious sequence of actions, which can not be statically checked. Another example shows that centralised runtime verfication can not ensure a sequence of actions meet the intended requirements of local processes due to the latency of routing.

Assume the global specification satisfying the consistency principles is given below:

$$\begin{array}{rcl} G_1 & = & p \to p_1 : (x : \mathsf{int})\langle\!\langle x > 0 \rangle\!\rangle.p_1 \to p_2 : (x : \mathsf{int})\langle\!\langle \mathsf{true} \rangle\!\rangle. \\ & & p \to p_2 : (y : \mathsf{int})\langle\!\langle y > x > 0 \rangle\!\rangle.\mathsf{end} \end{array}$$

applying the projection rules defined in Definition 6.5.1, the local specification of the process playing role p in session s is:

$$s[p] : p_1!(x : \mathsf{int})\langle\!\langle x > 0 \rangle\!\rangle.p_2!(y : \mathsf{int})\langle\!\langle y > x > 0 \rangle\!\rangle.\mathsf{end}$$

The global invariants of this local specification are (I) the possible sequence of actions of the local process playing $s[p]$ is either $s[p, p_1]!\langle v_1\rangle \cdot s[p, p_2]!\langle v_2\rangle$ or $s[p, p_2]!\langle v_2\rangle \cdot s[p, p_1]!\langle v_1\rangle$, and (II) the content of message sending to role p_2 should be bigger than the one to role p_1. Although this specification is statically valid according to well-formedness and well-assertedness, and it can be statically realised that there are two

such possible sequences of actions, it is not enough for safety assurance. When untrusted components may exist in the network, during runtime, as a monitor observes the trace $s[p, p_2]!\langle v_2\rangle \cdot s[p, p_1]!\langle v_1\rangle$ sent from $s[p]$, according to Definition 7.2.2 (defined later in Section 7.2.1.2), it is not valid because $v_2 > x > 0 \downarrow \mathsf{false}$ since x is empty when the monitor observes this action. A monitor should reject it or buffer the content of y, which is v_2, sent to p_2 and wait for the content of x to decide if the first coming interaction $s[p, p_1]!\langle v_2\rangle$ is valid. Otherwise, if this sequence of actions is sent from a malicious process and monitor does not take off it, it may cause damages to the processes playing role p_1 and p_2 due to the contents do not meet the agreed conditions.

The following example says that the process playing role p is feeding the process playing role q with numbers:

$$\begin{array}{rcl} G_2 & = & p \to q : (x : \mathsf{int})\langle\!\langle x > 0\rangle\!\rangle.p \to q : (y : \mathsf{int})\langle\!\langle \mathsf{true}\rangle\!\rangle. \\ & & p \to q : (z : \mathsf{int})\langle\!\langle z > max(x, y)\rangle\!\rangle.\mathsf{end} \end{array}$$

The local specification projected from G_2 on the endpoint playing role q, with the assumption that it joins session s', is

$$s'[q] : p?(x : \mathsf{int})\langle\!\langle x > 0\rangle\!\rangle.p?(y : \mathsf{int})\langle\!\langle \mathsf{true}\rangle\!\rangle.p?(z : \mathsf{int})\langle\!\langle z > max(x, y)\rangle\!\rangle.\mathsf{end}$$

Since the order of inputted numbers may affect the computation at the process playing role q, e.g. role q inputs data for real-time experiment, this scenario restricts the sequence of actions, each of which can not be permuted. A centralised verification is powerless in this case because even it approves a sequence of actions, e.g. which is

$$s'[p, q]?(3).s'[p, q]?(2).s'[p, q]?(5)$$

when these actions arrive at process playing role q, due to the possible routing latency, the sequence of actions may become

$$s'[p, q]?(5).s'[p, q]?(3).s'[p, q]?(2)$$

which should be rejected by the local monitor guarding the process playing role q. This example shows that, in distributed systems, centralised monitor is not reliable because some tasks can only be ensured at local monitors.

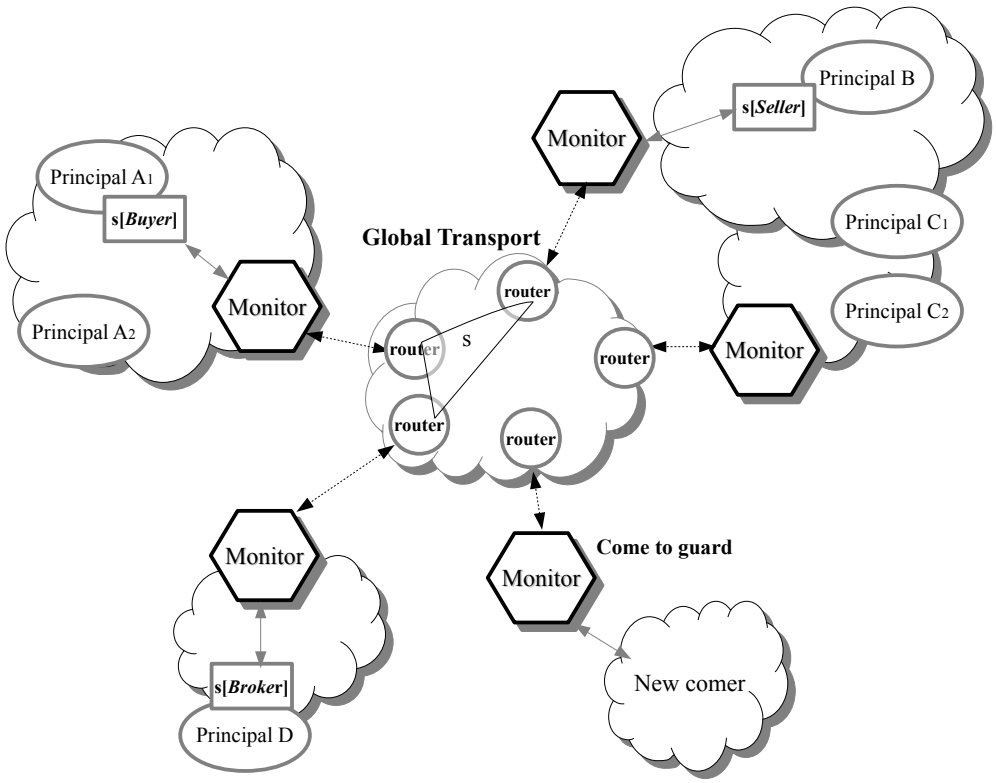

Figure 7.1: Monitoring Architecture

7.1.2 Monitoring Framework

We propose a monitoring framework as shown in Figure 7.1, which is motivated from the understanding of distributed infrastructures such as those for E-Science (129) and the infrastructure behind large-scale web portals. In this framework, each endpoint is linked to the network through an external monitor, which acts as a unique entry point to the infrastructure. In Figure 7.1, a monitor can guard more than one principal, who may have more than one process or endpoint. As a new comer joins the distributed systems, a monitor is generated by the system to guard it. When principals establish a session, e.g. Principals A_1, B, and D generate session s and play roles as $Buyer, Seller$ and $Broker$, corresponding local monitors guarding each domain inform this session creation information to distributed routers, which are responsible for global transport. In this thesis, the routers are not abstracted and formalised because our focus is on specifying specifications to verify interactions.

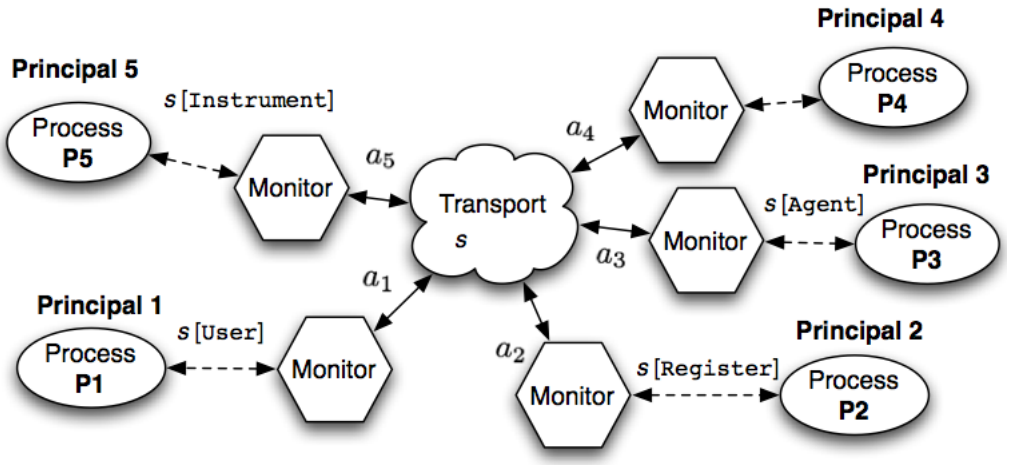

Figure 7.2: Monitoring for One Session s with the Specification: OO Instrument Commands

Figure 7.2 illustrates how global and local specifications are used by monitors to ensure the interactions of the use case of OOI Instrument Command (129) introduced in Chapter 5. In this scenario, there are five principals, each of which runs a process, involving in the network. To ensure global safety, monitors use projected endpoint specifications to detect whether or not *asynchronously* incoming and outgoing messages obey the stipulated specification, taking appropriate actions as necessary such as dropping illegal messages.

7.2 The Calculus of Monitoring

This section introduces a calculus for distributed monitoring, where every endpoint process is guarded by a corresponding system monitor. The syntax is divided into two parts: Section 7.2.1 for local monitors $\mathcal{M}$ which check the correctness of endpoints' behaviours through checking the incoming and outgoing messages (w.r.t. a set of endpoint specifications) and drop the wrong ones; and Section 7.2.4 for the distributed network N which consists of one or more monitored local processes $\mathcal{M}[P]$ and global queue containing pending messages sent by one monitored process but not yet received by another.

$$
\begin{array}{rcll}
\mathcal{M} & ::= & \Gamma, \Delta & \text{monitor} \\
\Delta & ::= & \emptyset \mid \Delta, s[p] : T \mid \Delta, s[p]^\bullet : T & \text{session environment} \\
\Gamma & ::= & \emptyset \mid \Gamma, a : \mathtt{mode}(G[p]) & \text{shared environment} \\
\mathtt{mode} & ::= & \mathtt{I} \mid \mathtt{O} \mid \mathtt{IO} & \text{I/O mode}
\end{array}
$$

Figure 7.3: Monitors ($\mathcal{M}$)

7.2.1 The Syntax and Semantics of Local External Monitors

Every process is associated with a corresponding external monitor that manages its interactions with the network (hence with other peers) by inspecting outgoing and incoming messages. In the following sections, we define the contents and the dynamic configurations of a monitor. A monitor uses these contents, which are specifications of local processes, as knowledge and follows the semantics of specifications to dynamically judge the behaviours of local processes.

7.2.1.1 The Syntax of Monitors

The syntax of monitors is defined in Figure 7.3. Γ is the same one defined in Definition 6.3.7. A monitor $\mathcal{M}$ checks two kinds of messages: (I) Capability exchanges at shared channels through invitations, and (II) session messages transmitted through session channels. The specification for a monitor consists of two specifying environments, one for shared channels (Γ) and one for sessions (Δ). In Δ, let $s[p] : T$ represent a capability which is owned by the local process but is not active yet (i.e., capability $s[p] : T$ can be sent to invite another process); while $s[p]^\bullet : T$ represents an active capability after the monitored process has joined the session with role p (i.e. playing session-role $s[p]$). Note that, as a process joins a session-role, say $s[p]^\bullet : T$, it can no more delegate this capability of playing $s[p]$ to another endpoint. Δ associates each capability, represented by $s[p]$, to an endpoint specification T which specifies the behaviour of a particular role in a session.

For a monitor $\mathcal{M} = \Gamma, \Delta$, to extract the elements governed by it, theoretically we can use $\mathsf{dom}(\Gamma)$ to extract all shared names $a, b, ...$ and use $\mathsf{dom}(\Delta)$ to extract all session-role $s[p], s'[q], ...$ which are governed by $\mathcal{M}$. They give us sufficient information for what endpoints that $\mathcal{M}$ governs: Shared names $a, b, ...$ with input mode $\mathtt{I}$, which represent endpoint's input communication channel, give us the positions of the endpoints; while

$s[p], s'[q], ...$ as session-roles give us which session participants are communicating to each other.

To runtime monitor non-deterministic sequences of actions, define the initial state of variables in a monitor as below:

Definition 7.2.1. Assume $\mathcal{M} = \Gamma, \Delta$. As an element of Γ or Δ is initiated in a monitor, the value of any variable, say x, occuring in this element is empty, denoted by $x \mapsto \emptyset$.

Note that, the elements of a monitor can be $\emptyset$, can have input mode shared names, e.g. $a : \mathtt{im}(G[p])$, or can have public session-roles, e.g. $s[p] : T$. When an endpoint process creates a session instance by doing $\overline{a}(s[p] : G)$ (see Section 5.3.1 in Chapter 5), its corresponding monitor learns that a new session is obeying global specification G. At the same time, it also learns the specification of $s[p]$ from the projection of G at role p.

The asynchronous transport between a process (as a session role $s[p]$) and its monitor is formalised by *(local) queue*, denoted by $s[p] : h$. The global transport among monitors for a session is formalised by *global queue*, denoted simply by H.

7.2.1.2 The Semantics of Monitors

The semantics of monitors show the dynamic configurations of monitors. It is given as a labelled transition system (LTS) following the standard interpretation of types in process calculi. This system is used to control behaviours of processes. $\mathcal{M} \xrightarrow{\ell} \mathcal{M}'$ only if ℓ is a legal action (i.e., that a process should be allowed to perform by monitor $\mathcal{M}$). The following definition is proposed for a monitor to correctly evaluate an assertion A.

Definition 7.2.2 (evaluable assertion).
Define A is evaluable whenever $\forall x \in var(A)$, x is not empty. When A is not evaluable, $A \downarrow \mathsf{false}$.

Figure 7.4 defines the LTS of monitors. Action labels are as same as those defined in Equation 5.1 in Chapter 5. Rule [New] approves an input-mode (i.e. mode of $\mathtt{im}$) shared name creation with specification $a : \mathtt{im}(G[p]), \mathtt{im} \in \{\mathtt{I}, \mathtt{IO}\}$ whenever name a is fresh to the monitor. Note that, as $a : \mathtt{im}(G[p])$ is fresh to a local monitor, this name is new to the network. The reason is explained in Convention 7.4.15.

[Tau] $$\mathcal{M} \xrightarrow{\tau} \mathcal{M}$$

[New] $$\frac{\mathtt{im} \in \{\mathtt{I}, \mathtt{IO}\}, a \notin \mathsf{dom}(\mathcal{M})}{\mathcal{M} \xrightarrow{\mathsf{new}\ a:\mathtt{im}(G[p])} \mathcal{M}, a : \mathtt{im}(G[p])}$$

[req-b] $$\frac{\begin{array}{c} s \notin \mathsf{dom}(\mathcal{M}),\ \mathtt{om} = \{\mathtt{O}, \mathtt{IO}\}, \\ \forall i \in I, p_i \in \mathit{role}(G),\ G \text{ is well-formed} \end{array}}{\mathcal{M}, a : \mathtt{om}(G[p_j]) \xrightarrow{\overline{a}(s[p_j]:G)} \mathcal{M}, a : \mathtt{om}(G[p_j]), \{s[p_i] : (G \upharpoonright p_i)\}_{i \in I \setminus \{j\}}}$$

[req-f] $$\frac{T = G \upharpoonright p_j,\ \mathtt{om} = \{\mathtt{O}, \mathtt{IO}\}}{\mathcal{M}, a : \mathtt{om}(G[p_j]), s[p_j] : T \xrightarrow{\overline{a}\langle s[p_j]:G\rangle} \mathcal{M}, a : \mathtt{om}(G[p_j])}$$

[Acc-b] $$\frac{s \notin \mathsf{dom}(\mathcal{M}),\ \mathtt{im} \in \{\mathtt{I}, \mathtt{IO}\}}{\mathcal{M}, a : \mathtt{im}(G[p]) \xrightarrow{a(s[p]:G)} \mathcal{M}, a : \mathtt{im}(G[p]), s[p] : T}$$

[Acc-f] $$\frac{s \in \mathsf{dom}(\mathcal{M}),\ \mathtt{im} \in \{\mathtt{I}, \mathtt{IO}\}}{\mathcal{M}, a : \mathtt{im}(G[p]) \xrightarrow{a\langle s[p]:G\rangle} \mathcal{M}, a : \mathtt{im}(G[p]), s[p] : T}$$

[Join] $$\mathcal{M}, s[p] : T \xrightarrow{\mathsf{join}(s[p])} \mathcal{M}, s[p]^\bullet : T$$

[Sel] $$\frac{\mathcal{M} \vdash v : S_j,\ A_j\{v/x_j\} \downarrow \mathsf{true},\ j \in I,\ T \curvearrowright p_2!\{l_i(x_i : S_i)\langle\!\langle A_i \rangle\!\rangle.T_i\}_{i \in I}}{\mathcal{M}, s[p_1]^\bullet : T \xrightarrow{s[p_1,p_2]!l_j\langle v\rangle} \mathcal{M}, s[p_1]^\bullet : T_j\{v/x_j\}}$$

[SelN] $$\frac{\mathcal{M} \vdash a : S_j, A_j \downarrow \mathsf{true},\ j \in I, T \curvearrowright p_2!\{l_i(x_i : S_i)\langle\!\langle A_i \rangle\!\rangle.T_i\}_{i \in I}}{\mathcal{M}, s[p_1]^\bullet : T \xrightarrow{s[p_1,p_2]!l_j\langle a\rangle} \mathcal{M}, s[p_1]^\bullet : T_j\{a/x_j\}}$$

[Bra] $$\frac{\mathcal{M} \vdash v : S_j,\ A_j\{v/x_j\} \downarrow \mathsf{true},\ j \in I,\ T \curvearrowright p_1?\{l_i(x_i : S_i)\langle\!\langle A_i \rangle\!\rangle.T_i\}_{i \in I}}{\mathcal{M}, s[p_2]^\bullet : T \xrightarrow{s[p_1,p_2]?l_j(v)} \mathcal{M}, s[p_2]^\bullet : T_j\{v/x_j\}}$$

[BraN] $$\frac{S_j = \mathsf{mode}(G'[p]), A_j \downarrow \mathsf{true},\ j \in I,\ T \curvearrowright p_1?\{l_i(x_i : S_i)\langle\!\langle A_i \rangle\!\rangle.T_i\}_{i \in I}}{\mathcal{M}, s[p_2]^\bullet : T \xrightarrow{s[p_1,p_2]?l_j(a)} \mathcal{M}, a : \mathtt{O}(G'[p]), s[p_2]^\bullet : T_j\{a/x_j\}}$$

[Par] $$\frac{\mathcal{M}_1 \xrightarrow{\ell} \mathcal{M}_1'}{\mathcal{M}_1 \parallel \mathcal{M}_2 \xrightarrow{\ell} \mathcal{M}_1' \parallel \mathcal{M}_2}$$

Figure 7.4: The Labelled Transition System of Monitors

Rule [Req-b] is for newing session s, which is specified under specification G, and at the same time inviting the endpoint represented by shared channel a to play session-role $s[p_j]$ specified under the local specification $G \upharpoonright p_j$. It proves the action $\overline{a}(s[p_j] : G)$ when s is fresh to the monitor $\mathcal{M}$ and $\mathcal{M}$ has the knowledge of $a : \mathtt{om}(G[p])$, $\mathtt{om} = \{\mathtt{O}, \mathtt{IO}\}$. Again, the local monitor can ensure this session name is globally new with the same reason explained for rule [New]. As the action is approved, it adds the projections from G on each role $p_i \in \mathit{role}(G)$, endowing the local process with the capability $s[p_i] : G \upharpoonright p_i$ of having behaviours of the role in G, and remove the capability of session-role $s[p_j] : T$, where $T = G \upharpoonright p_j$. Rule [Req-f] is for a free request at a. As a monitor has the knowledge of $a : \mathtt{om}(G[p])$ and the capability of $s[p_j] : T$, the monitor approves this invitation and, since this capability has been sent out, relinquishes the capability of $s[p_j] : T$; rule [Acc-f] is its dual by asking the capability of $a : \mathtt{im}(G[p]), \mathtt{im} \in \{\mathtt{I}, \mathtt{IO}\}$. Rule [Acc-b] is very similar to rule [Acc-f] except that [Acc-b] requires that session s is fresh to $\mathcal{M}$. Rule [Join] activates the capability of $s[p]$, denoted by $s[p]^\bullet$. Only as a capability is denoted by $s[p]^\bullet$, it fires and executes actions. In [Sel], [SelN], [Bra] and [BraN], *permutation* mechanism is applied, denoted by $T \curvearrowright T'$. A sound permutation changes the order of actions to capture the semantics of asynchronously arriving messages without affecting causally related actions. Consider a well-formed global specification

$$p_2 \to p_1 : (x\!:\!S)\langle\!\langle A \rangle\!\rangle.p_3 \to p_1 : (x'\!:\!S')\langle\!\langle A' \rangle\!\rangle.G,\ p_1 \neq p_2 \neq p_3$$

In an asynchronous environment, the message carrying x' may arrive at p_1 *earlier* than the message carrying x. Naturally, if A and A' have no causal relations in x and x', p_1's monitor should be able to accept the messages from p_2 and p_3 in any order. $T \curvearrowright T'$ is therefore useful for helping a monitor to evaluate message passing.

With the mechanism of permutations, [Sel] says that, if the local specification of $s[p_1]^\bullet$ can be permuted to $p_2!\{l_i(x_i\!:\!S_i)\langle\!\langle A_i \rangle\!\rangle.T_i\}_{i\in I}$, the outgoing message with label l_j sent from p_1 to p_2, as far as it satisfies formula $A_j\{v/x_j\}$, is approved by the monitor which prepares the next (incoming or outgoing) message with local specification T_j. [SelN] is similar to [Sel] except that it is particularly for sending a shared name a. Note that A_j should be always true since a name can not be evaluated in a predicate. Rule [Bra] is the symmetric (input) counterpart to [Sel], while [BraN] is the rule for receiving a shared name a. It is dual to the rule [SelN].

Note that, in [BRAN], a is added into a monitor with `O`-mode no matter what the original `mode` of a is, i.e. it is never added with `I`-mode or `IO`-mode. Together with the rule [SELN], they imply that, when a shared name is passed from a sender to a receiver through session interactions, the sender does not lose the capability/knowledge of the shared name with its original I/O mode. Therefore, passing a name in session interactions is a propagation of the knowledge of the name. As Section 5.5.1 explained, the mobility is realised by passing session-roles but not by passing input-mode shared names.

Rule [PAR] says that, when a monitor, say $\mathcal{M}_1$, approves an action and becomes $\mathcal{M}_1'$, this action is network-wise unique such that no other monitors observe this action at the same time because the session contained in this action is unique to the network.

The next section follows the illustration of the LTS of monitor-off $\mathcal{M}^\circ$, which can be viewed as gateways. Its formalism helps us to deepen the understanding about how floating messages can be routed from one endpoint to another. Compared with the semantics of monitor-off, we can realise the effects of monitors for guarding interactions.

7.2.2 The Syntax and Semantics of Monitor-off Gateway

Unmonitored networks are given by erasing the co-domain (sorts and specifications) from each monitor in a monitored network. The result of such erasure, the monitor which 'switches-off' the monitoring activity, acts simply as a *gateway* with information on local addresses, including a session endpoint, say $s[p]$. Write $\mathsf{erase}(\mathcal{M})$ for the monitor-off obtained through this erasure of $\mathcal{M}$. Hereafter denote $\mathsf{erase}(\mathcal{M})$ as $\mathcal{M}^\circ$. This stripped-off $\mathcal{M}$ is called a *monitor-off gateway* $(\mathcal{M}^\circ, \mathcal{M}_1^\circ, \ldots)$. The syntax of monitor-off gateways is defined as follows:

$$\mathcal{M}^\circ ::= \emptyset \mid \mathcal{M}^\circ, a \mid \mathcal{M}^\circ, s[p] \mid \mathcal{M}^\circ, s[p]^\bullet$$

A gateway $\mathcal{M}^\circ$ is simply defined as removing the typing area from a $\mathcal{M}$.

The LTS of $\mathcal{M}^\circ$ is obtained by erasing the co-domain from each monitor in each rule in Figure 7.4. Below $\mathcal{M}^\circ, a$ indicates $a \notin \mathcal{M}^\circ$, similarly for $\mathcal{M}^\circ, s[p]$. The illustrations of each rules are as followings:

[$\mathcal{M}^\circ$-tau] $\mathcal{M}^\circ$ does nothing to a silent action τ.

$$\mathcal{M}^\circ \xrightarrow{\tau} \mathcal{M}^\circ \qquad \mathcal{M}^\circ \xrightarrow{\text{new } a:\mathtt{mode}(G[p])} \mathcal{M}^\circ, a \qquad [\mathcal{M}^\circ\text{-tau}, \mathcal{M}^\circ\text{-new a}]$$

$$\mathcal{M}^\circ \xrightarrow{\overline{a}(s[p_j]:G)} \mathcal{M}^\circ, \{s[p_i]\}_{i\in I\setminus\{j\}} \quad \mathcal{M}^\circ, a, s[p_j] \xrightarrow{\overline{a}\langle s[p_j]:G\rangle} \mathcal{M}^\circ, a \qquad [\mathcal{M}^\circ\text{-request b/f}]$$

$$\mathcal{M}^\circ, a \xrightarrow{a(s[p]:G)} \mathcal{M}^\circ, a, s[p] \quad \mathcal{M}^\circ, a \xrightarrow{a\langle s[p]:G\rangle} \mathcal{M}^\circ, a, s[p] \qquad [\mathcal{M}^\circ\text{-acc b/f}]$$

$$\mathcal{M}^\circ, s[p] \xrightarrow{\mathsf{join}(s[p])} \mathcal{M}^\circ, s[p]^\bullet \qquad [\mathcal{M}^\circ\text{-join}]$$

$$\mathcal{M}^\circ, s[p_1]^\bullet \xrightarrow{s[p_1,p_2]!l\langle v\rangle} \mathcal{M}^\circ, s[p_1]^\bullet \quad \mathcal{M}^\circ, s[p_1]^\bullet \xrightarrow{s[p_1,p_2]!l\langle a\rangle} \mathcal{M}^\circ, s[p_1]^\bullet \qquad [\mathcal{M}^\circ\text{-sel}, \mathcal{M}^\circ\text{-selN}]$$

$$\mathcal{M}^\circ, s[p_2]^\bullet \xrightarrow{s[p_1,p_2]?l(v)} \mathcal{M}^\circ, s[p_2]^\bullet \qquad [\mathcal{M}^\circ\text{-bra}]$$

$$\mathcal{M}^\circ, s[p_2]^\bullet \xrightarrow{s[p_1,p_2]?l(a)} (\mathcal{M}^\circ, s[p_2]^\bullet), a \qquad [\mathcal{M}^\circ\text{-braN}]$$

Figure 7.5: The Labelled Transition System of Monitor-off

[$\mathcal{M}^\circ$-new a] $\mathcal{M}^\circ$ unions a name a no matter it is "new " to $\mathcal{M}^\circ$ or not. Because there is no monitor to specify the newed shared name is input-mode, even the mode is an output mode, $\mathcal{M}^\circ$ also accepts the action. Thus `mode` is used here instead of using `im`.

[$\mathcal{M}^\circ$-req b/f] By rule [$\mathcal{M}^\circ$-req b], $\mathcal{M}^\circ$ adds all session-roles of session s whether session s is new to $\mathcal{M}^\circ$ or not. $\mathcal{M}^\circ$ does not know any specification of capability of the roles in $\{s[p_i]\}_{i\in I\setminus\{j\}}$. Rule [$\mathcal{M}^\circ$-req f] says $s[p_j]$ is removed from $\mathcal{M}^\circ$ via channel a when request happens.

[$\mathcal{M}^\circ$-acc b/f] These two rules have same effects for two different actions $a(s[p] : G)$ and $a\langle s[p] : G\rangle$. $s[p]$ is added into $\mathcal{M}^\circ$ through channel a with no conditions.

[$\mathcal{M}^\circ$-join] $\mathcal{M}^\circ$ activates the capability of role p in session s. $\mathcal{M}^\circ$ announces it to the network that role p in session s is ready to start session interactions.

[$\mathcal{M}^\circ$-sel, -selN] Rule [$\mathcal{M}^\circ$-sel] and [$\mathcal{M}^\circ$-selN] both say $\mathcal{M}^\circ$ allows $s[p_1, p_2]!l\langle v\rangle$ or $s[p_1, p_2]!l\langle a\rangle$ when its endpoint is the sender p_1 in s.

[$\mathcal{M}^\circ$-bra] As v is not a name, and $\mathcal{M}^\circ$'s endpoint is the receiver p_2 in s, it approves.

[$\mathcal{M}^\circ$-braN] As $\mathcal{M}^\circ$'s endpoint is the receiver p_2 in s, it approves it and adds a into $\mathcal{M}^\circ$.

Example 7.2.3 ($\mathcal{M}$ v.s. $\mathcal{M}^\circ$). Let $\mathcal{M}_1 \stackrel{\text{def}}{=} s[p_1]^\bullet : p_2!(x : \mathsf{Int})\langle\!\langle x > 0 \rangle\!\rangle.T$ and $\ell = s[p_1, p_2]!\langle -10 \rangle$. Then $\mathcal{M}_1^\circ \xrightarrow{\ell}$ but $\mathcal{M}_1 \not\xrightarrow{\ell}$ since the value -10 does not satisfy the predicate $x > 0$ attached to the endpoint specification for the session monitored by $\mathcal{M}_1$. Similarly, for specification violations, if $\mathcal{M}_2 \stackrel{\text{def}}{=} s[p_1]^\bullet : p_2!(x\!:\!\mathsf{String})\langle\!\langle \mathsf{true} \rangle\!\rangle.T$ and $\ell = s[p_1, p_2]!\langle 10 \rangle$ then $\mathcal{M}_2^\circ \xrightarrow{\ell}$ but $\mathcal{M}_2 \not\xrightarrow{\ell}$.

7.2.3 The Syntax and Semantics of Monitored Processes

Write $\mathcal{M}[P]$ for P monitored by $\mathcal{M}$, called *monitored process*. The syntax of a monitored process is defined as a local process guarded by an external monitor.

Definition 7.2.4. Define $\mathcal{M}[P]$ as a monitored local process.

Definition 7.2.5. $\mathcal{M}[\mathbf{0}] \equiv \mathbf{0}$.

The semantics of monitored processes is simply defined based on the semantics of local processes and monitors:

Definition 7.2.6 (LTS of monitored process). Let ℓ be an action label. Then we define $\mathcal{M}_1[P_1] \xrightarrow{\ell} \mathcal{M}_2[P_2]$ as the conjunction of $P_1 \xrightarrow{\ell} P_2$ and $\mathcal{M}_1 \xrightarrow{\ell} \mathcal{M}_2$:

$$\frac{\mathcal{M}_1 \xrightarrow{\ell} \mathcal{M}_2, P_1 \xrightarrow{\ell} P_2}{\mathcal{M}_1[P_1] \xrightarrow{\ell} \mathcal{M}_2[P_2]}$$

The LTS of P and $\mathcal{M}$ are respectively defined in Figure 5.4 (Chapter 5) and Figure 7.4. A local queue of a local process is abstracted for asynchronous monitoring purpose. During runtime, a local queue is part of a runtime process. Formally, we write

$$\mathcal{M}[P_{rt}] = \mathcal{M}[P \mid s_{i_1}[p_j] : h_{i_1} \mid \ldots \mid s_{i_n}[p_k] : h_{i_n}]$$

where $s_{i_1}[p_j] : h_{i_1} \mid \ldots \mid s_{i_n}[p_k] : h_{i_n}$ are the local queues generated from participating sessions $s_{i_1}, ..., s_{i_n}$ respectively for playing roles $p_j, ..., p_k$.

Example 7.2.7. When a process, say P, involves in a session s for playing role p, during runtime, its runtime monitored process is

$$\mathcal{M}[P_{rt}] = \mathcal{M}[P \mid s[p] : h]$$

$$
\begin{array}{rcll}
N_s & ::= & \mathcal{M} \mid \mathcal{M}[P] \mid N_{s1} \parallel N_{s2} \mid \mathbf{0} \mid (\nu n)N_s \; n \in \{a, s\} & \text{monitored static network} \\
N & ::= & N_s \mid N_1 \parallel N_2 \mid H \mid (\nu n)N \; n \in \{a, s\} & \text{monitored dynamic network} \\
H & ::= & \emptyset \mid m \cdot H & \text{global queue} \\
m & ::= & s\langle p, q, l\langle v\rangle\rangle \mid \overline{a}\langle s[p] : G\rangle & \text{message}
\end{array}
$$

Figure 7.6: The Syntax of the Monitored Network

where $\mathcal{M}$ should contain the local specification of $s[p]$ such that $\mathcal{M} = \Gamma, \Delta, s[p] : T$.

Example 7.2.8. Let $P = s[p_1, p_2]!\langle 100\rangle; P' \mid s[p_1] : \emptyset$ be a process with an empty queue playing role p_1 in s. Also let $\mathcal{M}_1 = s[p_1]^\bullet : p_2!(x : \mathsf{Int})\langle\!\langle x > 0\rangle\!\rangle.T$ be its local monitor. The communication happens in two steps. First, the message is spawn into the local queue as $P \xrightarrow{\tau} P_2$ with $P_2 = P' \mid s[p_1] : s\langle p_1, p_2, \langle 100\rangle\rangle$ hence, since $\mathcal{M}_1 \xrightarrow{\tau} \mathcal{M}_1$, then $\mathcal{M}_1[P] \xrightarrow{\tau} \mathcal{M}_1[P_2]$. Second, the message is forwarded to the global queue as $P_2 \xrightarrow{\ell} P' \mid s[p_1] : \emptyset$ with $\ell = s[p_1, p_2]!\langle 100\rangle$; hence since $\mathcal{M}_1 \xrightarrow{\ell} s[p_1]^\bullet : T$ then $\mathcal{M}_1[P] \xrightarrow{\ell} (s[p_1]^\bullet : T)[P' \mid s[p_1] : \emptyset]$.

7.2.4 The Syntax and Semantics of the Monitored Network

Based on the syntax and semantics of the network and monitors introduced in Chapter 5 and Section 7.2.1, this section introduces the calculus of the monitored network.

7.2.4.1 The Syntax of the Monitored Network

The syntax of the monitored network is similar to the one of the un-monitored network but with notation N rather than N (i.e. un-monitored network), which is defined in Figure 5.5, Chapter 5. The syntax of the *monitored network* $(N, N', \ldots)$ is defined in Figure 7.6. The following definitions are for illustrating the syntax of the monitored network defined in Figure 7.6.

Definition 7.2.9 (Monitored Network). Assume a full (dynamic) network N is given. We say N is a monitored network coming from N if and only if $\forall P \in \mathsf{N}$, P is guarded by a monitor $\mathcal{M} \in N$ such that $\mathcal{M}[P] \in N$.

Based on Definition 7.2.9, we define a monitored process more precisely:

Definition 7.2.10. Assume N is a monitored network coming from N. $\mathcal{M}[P] \in N$ if and only if $\mathsf{N} \equiv_{\mathsf{N}} (\nu\tilde{n})(M[P] \parallel \mathsf{N}')$ for some names $\tilde{n}$.

The structural congruency over the monitored network is defined as same as the one of the network (see Definition 5.4.2).

The composability and the parallel composition of two monitored networks are defined as same as those defined in Definitions 5.4.2 and 5.4.4. A monitored network represents a network of processes and their monitors, together with messages in transit: They are interaction message, i.e. $s\langle p, q, l\langle v\rangle\rangle$, and invitation, i.e. $\overline{a}\langle s[p] : G\rangle$. Both $s\langle p, q, l\langle v\rangle\rangle$ and $\overline{a}\langle s[p] : G\rangle$ are introduced in Figure 5.5, Chapter 5. Note that, without loss of generality, the global queue is shared by all processes participating in the network.

7.2.4.2 The Reduction Rules of the Monitored Network

Before introducing the reduction rules of the monitored network, one observation is noted:

Remark 7.2.11. The labelled transition systems (LTS) of local *monitored processes*, which are parts of the monitored network, from the global viewpoint, are reductions rules of the *monitored network*.

Globally, the actions happening at local processes are marginal, which are network-wise *invisible*; the visible actions can be and can only be observed by the global transport. Therefore, the actions happening in Figure 7.4 are globally invisible: they are reductions of the network. This observation corresponds to Remark 5.4.7 in Chapter 5.

The reduction rules of the monitored network is defined in Figure 7.7. In each rule except ⌊Proc⌋, the (else) case is when the monitor detects a violation w.r.t. specifications by the current action; for this case, $\mathcal{M}$ simply drops such message. Each rule corresponds to a rule of monitor semantics defined in Figure 7.4. Rule ⌊New⌋ creates a fresh (bound) input-mode shared name, where $\texttt{im} \in \{\texttt{I}, \texttt{IO}\}$. In rules ⌊Req-b/f -out, Acc-b/f -in⌋, monitor $\mathcal{M}$ checks outgoing and incoming invitations. ⌊Req-b/f -out⌋ forwards outgoing invitations, which are still local to a process, to the external environment (dually for ⌊Acc-b/f -in⌋ with incoming invitations). Particularly, rules ⌊Req-b-out⌋ and ⌊Acc-b-in⌋ also check if session s is new to the endpoint process. Note that ⌊Req-b-out⌋ defines that, as error happens, the process following the bound request $\overline{a}(s[p] : G)$ is completely taken off:

$$\mathcal{M}[\overline{a}(s[p] : G).P] \parallel H \longrightarrow \mathcal{M}[\mathbf{0}] \parallel H$$

$$
\begin{array}{lrcll}
\lfloor\textsc{Proc}\rfloor & \mathcal{M}[P] & \longrightarrow & \mathcal{M}[P'] & (P \xrightarrow{\tau} P') \\
\lfloor\textsc{New-a}\rfloor & \mathcal{M}[\mathsf{new}\ a : \mathtt{im}(G[p]).P] & \longrightarrow & \begin{cases}(\nu a)(\mathcal{M}'[P]) \\ \mathcal{M}[\mathbf{0}]\end{cases} & \begin{array}{l}(\mathcal{M} \xrightarrow{\mathsf{new}\ a:\mathtt{im}(G[p])} \mathcal{M}') \\ (\text{else})\end{array} \\
\lfloor\textsc{Req-b-out}\rfloor & \mathcal{M}[\overline{a}(s[p] : G).P] \parallel H & \longrightarrow & \begin{cases}(\nu s)(\mathcal{M}'[P] \parallel H \cdot \overline{a}\langle s[p] : G\rangle) \\ \mathcal{M}[\mathbf{0}] \parallel H\end{cases} & \begin{array}{l}(\mathcal{M} \xrightarrow{\overline{a}(s[p]:G)} \mathcal{M}') \\ (\text{else})\end{array} \\
\lfloor\textsc{Req-f-out}\rfloor & \mathcal{M}[\overline{a}\langle s[p] : G\rangle] \parallel H & \longrightarrow & \begin{cases}\mathcal{M}'[\mathbf{0}] \parallel H \cdot \overline{a}\langle s[p] : G\rangle \\ \mathcal{M}[\mathbf{0}] \parallel H\end{cases} & \begin{array}{l}(\mathcal{M} \xrightarrow{\overline{a}\langle s[p]:G\rangle} \mathcal{M}') \\ (\text{else})\end{array} \\
\lfloor\textsc{Acc-b-in}\rfloor & \mathcal{M}[\mathbf{0}] \parallel \overline{a}\langle s[p] : G\rangle \cdot H & \longrightarrow & \begin{cases}\mathcal{M}'[\overline{a}\langle s[p] : G\rangle] \parallel H \\ \mathcal{M}[\mathbf{0}] \parallel H\end{cases} & \begin{array}{l}(\mathcal{M} \xrightarrow{a(s[p]:G)} \mathcal{M}') \\ (\text{else})\end{array} \\
\lfloor\textsc{Acc-f-in}\rfloor & \mathcal{M}[\mathbf{0}] \parallel \overline{a}\langle s[p] : G\rangle \cdot H & \longrightarrow & \begin{cases}\mathcal{M}'[\overline{a}\langle s[p] : G\rangle] \parallel H \\ \mathcal{M}[\mathbf{0}] \parallel H\end{cases} & \begin{array}{l}(\mathcal{M} \xrightarrow{a\langle s[p]:G\rangle} \mathcal{M}') \\ (\text{else})\end{array} \\
\lfloor\textsc{Join}\rfloor & \mathcal{M}[\mathsf{join}(s[p]).P] & \longrightarrow & \begin{cases}\mathcal{M}'[P \mid s[p]{:}\emptyset] \\ \mathcal{M}[P]\end{cases} & \begin{array}{l}(\mathcal{M} \xrightarrow{\mathsf{join}(s[p])} \mathcal{M}') \\ (\text{else})\end{array} \\
\lfloor\textsc{Sel-out}\rfloor & \mathcal{M}[s[p_1] : s\langle p_1, p_2, l\langle v\rangle\rangle \cdot h] \parallel H & \longrightarrow & \begin{cases}\mathcal{M}'[s[p_1] : h] \parallel H \cdot s\langle p_1, p_2, l\langle v\rangle\rangle \\ \mathcal{M}[s[p_1] : h] \parallel H\end{cases} & \begin{array}{l}(\mathcal{M} \xrightarrow{s[p_1,p_2]!l\langle v\rangle} \mathcal{M}') \\ (\text{else})\end{array} \\
\lfloor\textsc{Bra-in}\rfloor & \mathcal{M}[s[p_2] : h] \parallel s\langle p_1, p_2, l\langle v\rangle\rangle \cdot H & \longrightarrow & \begin{cases}\mathcal{M}'[s[p_2] : h \cdot s\langle p_1, p_2, l\langle v\rangle\rangle] \parallel H \\ \mathcal{M}[s[p_2] : h] \parallel H\end{cases} & \begin{array}{l}(\mathcal{M} \xrightarrow{s[p_1,p_2]?l\langle v\rangle} \mathcal{M}') \\ (\text{else})\end{array}
\end{array}
$$

$$
\lfloor\textsc{par, res, str}\rfloor \quad \frac{N_1 \longrightarrow N_1'}{N_1 \parallel N_2 \longrightarrow N_1' \parallel N_2} \qquad \frac{N \longrightarrow N'}{(\nu\tilde{n})N \longrightarrow (\nu\tilde{n})N'} \qquad \frac{N \equiv_N N_0 \quad N_0 \longrightarrow N_0' \quad N_0' \equiv_N N'}{N \longrightarrow N'}
$$

Figure 7.7: The Reduction Rules of the Monitored Network

It is because the following process may contain session s which is going to be created by $\overline{a}(s[p] : G)$ (refer to the exmple in Section 5.2.3 in Chapter 5). Rule $\lfloor\textsc{Join}\rfloor$ creates a local queue particularly for session-role $s[p]$. In $\lfloor\textsc{Sel-out}\rfloor$, $\mathcal{M}$ forwards a session message from a local queue $s[p]$ to the global queue (dually for $\lfloor\textsc{Bra-in}\rfloor$). Other rules are standard.

7.2.4.3 The Labelled Tansition System of the Monitored Network

As $\xrightarrow{\ell}_{\mathrm{g}}$ is particularly defined for the LTS of the global transport (see Figure 5.7) to indicate that these actions are globally observable actions, it is also used for the LTS of the monitored network in Figure 7.8 for representing that the transitions are globally observable up to monitoring. Thus we use $\xrightarrow{\ell}_{\mathrm{g}}$ to describe the actions which can be observed by monitors and can *affect the overall network*. $N \xrightarrow{\ell}_{\mathrm{g}} N'$ represents the global observability of the network. Although, as mentioned before, the local actions are

invisible to the network, we can apply monitors's knowledge to describe these invisible actions (w.r.t the network) as visible ones by using $\xrightarrow{\ell}_{\mathrm{g}}$ because these actions actually have been observed by monitors.

Remark 7.2.12. A monitor, which positions between local processes and the network, is a *bridge* in between.

Therefore, global observability is the aggregate of what all monitors observe.

Definition 7.2.13 (global observable transition). Assume N is a monitored network. $N \xrightarrow{\ell}_{\mathrm{g}} N'$ if and only there exists $\mathcal{M} \in N$ s.t. $\mathcal{M} \xrightarrow{\ell} \mathcal{M}'$, $\mathcal{M}' \in N'$.

Each rule in Figure 7.7 comes from the rule of LTS of monitored processes, in which a corresponding local monitor approves an action. The actions of monitored processes are globally invisible since they are marginal actions to the network. The dynamics of the monitored network, in particular how messages travel from an endpoint through the network to another, is formalised by the LTS defined in Figure 7.8. In all rules, we assume processes and the network obey the standard binding convention.

The rules in Figure 7.8 are dual to those in Figure 5.7 for the LTS of the global transport. In other words, the action which can be observed through the global transport is the duality of the observable action in the monitored network. Figure 7.8 lists the rules when the actions are valid, i.e. those have been approved by monitors. As the actions are invalid, as those shown in Figure 7.9, if they are output actions, they should be intercepted as they are still at the local; if they are input actions, they should be intercepted as they are still in the global queue. In other words, the reduction rules in Figure 7.7 have concluded every case.

All action rules defined in reduction rules of the monitored network are known by the monitors, and therefore are informed to the network by using $\xrightarrow{\ell}_{\mathrm{g}}$. The illustrations of the LTS of the monitored network are as followings.

1. Rules ⌊MG-REQ-F⌋, ⌊MG-ACC-B⌋, ⌊MG-ACC-F⌋, ⌊MG-SEL⌋, ⌊MG-BRA⌋, and ⌊MG-IF-T/F⌋ just re-state the invisible action occuring at the local process. Here we particularly explain rule ⌊MG-ACC-F⌋: When $\overline{a}\langle s[p] : G\rangle$ enters the local process and is with the process in parallel, it must have been approved by the monitor; in other words, we should have $a : \mathtt{im}(G[p]) \in \mathcal{M}$, $\mathtt{im} \in \{\mathtt{I}, \mathtt{IO}\}$, where a is able to play role p specified under G. Therefore, $\overline{a}\langle s[p] : G\rangle \mid a\langle s[p] : G\rangle; P$ is a pattern-matching, which has already been checked by rule ⌊MG-ACC-F-IN⌋ and will not affect process P.

$$\frac{\mathtt{im} \in \{\mathtt{I}, \mathtt{IO}\}, \mathcal{M} \xrightarrow{\mathsf{new}\ a:\mathtt{im}(G[p])} \mathcal{M}'}{\mathcal{M}[\mathsf{new}\ a : \mathtt{im}(G[p]).P] \parallel H \xrightarrow{\mathsf{new}\ a:\mathtt{im}(G[p])}_{\mathrm{g}} (\nu a)(\mathcal{M}'[P]) \parallel H} \quad \lfloor \text{MG-NEW-A} \rfloor$$

$$\frac{\mathcal{M} \xrightarrow{\mathsf{join}(s[p])} \mathcal{M}'}{\mathcal{M}[\mathsf{join}(s[p]).P] \xrightarrow{\mathsf{join}(s[p])}_{\mathrm{g}} \mathcal{M}'[P \mid s[p]{:}\emptyset]} \quad \lfloor \text{MG-JOIN} \rfloor$$

$$\mathcal{M}[\overline{a}\langle s[p] : G\rangle; P] \xrightarrow{\tau}_{\mathrm{g}} \mathcal{M}[\overline{a}\langle s[p] : G\rangle \mid P] \quad \lfloor \text{MG-REQ-F} \rfloor$$

$$\frac{\mathcal{M} \xrightarrow{\overline{a}(s[p]:G)} \mathcal{M}'}{\mathcal{M}[\overline{a}(s[p] : G).P] \parallel H \xrightarrow{\overline{a}(s[p]:G)}_{\mathrm{g}} (\nu s)(\mathcal{M}'[P] \parallel H \cdot \overline{a}\langle s[p] : G\rangle)} \quad \lfloor \text{MG-REQ-B-OUT} \rfloor$$

$$\frac{\mathcal{M} \xrightarrow{\overline{a}\langle s[p]:G\rangle} \mathcal{M}'}{\mathcal{M}[\overline{a}\langle s[p] : G\rangle] \parallel H \xrightarrow{\overline{a}\langle s[p]:G\rangle}_{\mathrm{g}} \mathcal{M}'[\mathbf{0}] \parallel H \cdot \overline{a}\langle s[p] : G\rangle} \quad \lfloor \text{MG-REQ-F-OUT} \rfloor$$

$$\mathcal{M}[\overline{a}\langle s[p] : G\rangle \mid a(y[p] : G).P] \xrightarrow{\tau}_{\mathrm{g}} \mathcal{M}[P\{s/y\}] \quad \lfloor \text{MG-ACC-B} \rfloor$$

$$\mathcal{M}[\overline{a}\langle s[p] : G\rangle \mid a\langle s[p] : G\rangle; P] \xrightarrow{\tau}_{\mathrm{g}} \mathcal{M}[P] \quad \lfloor \text{MG-ACC-F} \rfloor$$

$$\frac{\mathcal{M} \xrightarrow{a(s[p]:G)} \mathcal{M}'}{\mathcal{M}[P] \parallel \overline{a}\langle s[p] : G\rangle \cdot H \xrightarrow{a(s[p]:G)}_{\mathrm{g}} \mathcal{M}'[P \mid \overline{a}\langle s[p] : G\rangle] \parallel H} \quad \lfloor \text{MG-ACC-B-IN} \rfloor$$

$$\frac{\mathcal{M} \xrightarrow{a\langle s[p]:G\rangle} \mathcal{M}'}{\mathcal{M}[P] \parallel \overline{a}\langle s[p] : G\rangle \cdot H \xrightarrow{a\langle s[p]:G\rangle}_{\mathrm{g}} \mathcal{M}'[P \mid \overline{a}\langle s[p] : G\rangle] \parallel H} \quad \lfloor \text{MG-ACC-F-IN} \rfloor$$

$$\mathcal{M}[s[p_1, p_2]!l_j\langle v\rangle; P \mid s[p_1]{:}h] \xrightarrow{\tau}_{\mathrm{g}} \mathcal{M}[P \mid s[p_1]{:}h \cdot s\langle p_1, p_2, l_j\langle v\rangle\rangle] \quad \lfloor \text{MG-SEL} \rfloor$$

$$\mathcal{M}[s[p_1, p_2]?\{l_i(x_i).P_i\}_{i\in I} \mid s[p_2]{:}s\langle p_1, p_2, l_j\langle v\rangle\rangle \cdot h] \xrightarrow{\tau}_{\mathrm{g}} \mathcal{M}[P_j\{v/x_j\} \mid s[p_2]{:}h] \quad \lfloor \text{MG-BRA} \rfloor$$

$$\frac{\mathcal{M} \xrightarrow{s[p_1,p_2]!l\langle v\rangle} \mathcal{M}'}{\mathcal{M}[P \mid s[p_1]{:}s\langle p_1, p_2, l\langle v\rangle\rangle \cdot h] \parallel H \xrightarrow{s[p_1,p_2]!l\langle v\rangle}_{\mathrm{g}} \mathcal{M}'[P \mid s[p_1]{:}h] \parallel H \cdot s\langle p_1, p_2, l\langle v\rangle\rangle} \quad \lfloor \text{MG-SEL-OUT} \rfloor$$

$$\frac{\mathcal{M} \xrightarrow{s[p_1,p_2]?l(v)} \mathcal{M}'}{\mathcal{M}[P \mid s[p_2]{:}h] \mid s\langle p_1, p_2, l\langle v\rangle\rangle \cdot H \xrightarrow{s[p_1,p_2]?l(v)}_{\mathrm{g}} \mathcal{M}'[P \mid s[p_2]{:}h \cdot s\langle p_1, p_2, l\langle v\rangle\rangle] \parallel H} \quad \lfloor \text{MG-BRA-IN} \rfloor$$

$$\mathcal{M}[\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ P\ \mathsf{else}\ Q] \xrightarrow{\tau}_{\mathrm{g}} \mathcal{M}[P] \quad \mathcal{M}[\mathsf{if}\ \mathsf{false}\ \mathsf{then}\ P\ \mathsf{else}\ Q] \xrightarrow{\tau}_{\mathrm{g}} \mathcal{M}[Q] \quad \lfloor \text{MG-IF-T/F} \rfloor$$

$$\frac{N_1 \xrightarrow{\ell}_{\mathrm{g}} N_1' \quad \mathsf{bn}(\ell) \cap \mathsf{fn}(N_2) = \emptyset \quad \mathsf{dest}(\ell) \notin \mathsf{ep}(N_2)}{N_1 \parallel N_2 \xrightarrow{\ell}_{\mathrm{g}} N_1' \parallel N_2} \quad \lfloor \text{MG-PAR} \rfloor$$

$$\frac{N \xrightarrow{\ell}_{\mathrm{g}} N' \quad \mathsf{n}(\ell) \notin \tilde{n}}{(\nu\tilde{n})N \xrightarrow{\ell}_{\mathrm{g}} (\nu\tilde{n})N'} \quad \frac{N_1 \equiv_N N_2 \xrightarrow{\ell}_{\mathrm{g}} N_2' \equiv_N N_1'}{N_1 \xrightarrow{\ell}_{\mathrm{g}} N_1'} \quad \lfloor \text{MG-RES,STR} \rfloor$$

Figure 7.8: The Labelled Transition System of the Monitored Network as Actions are Valid

$$\frac{\mathtt{im} \in \{\mathtt{I}, \mathtt{IO}\}, \mathcal{M} \xrightarrow{\mathsf{new}\ a:\mathtt{im}(G[p])} \!\!\!\!\!\!/\ \quad}{\mathcal{M}[\mathsf{new}\ a : \mathtt{im}(G[p]).P] \xrightarrow{\tau}_{\mathrm{g}} \mathcal{M}[P]} \quad \lfloor \text{MG-NEW-A-ERR} \rfloor$$

$$\frac{\mathcal{M} \xrightarrow{\overline{a}(s[p]:G)} \!\!\!\!\!\!/\ \quad}{\mathcal{M}[\overline{a}(s[p] : G).P] \xrightarrow{\tau}_{\mathrm{g}} \mathcal{M}[\mathbf{0}]} \quad \lfloor \text{MG-REQ-B-OUT-ERR} \rfloor$$

$$\frac{\mathcal{M} \xrightarrow{\overline{a}\langle s[p]:G\rangle} \!\!\!\!\!\!/\ \quad}{\mathcal{M}[\overline{a}\langle s[p] : G\rangle] \xrightarrow{\tau}_{\mathrm{g}} \mathcal{M}[\mathbf{0}]} \quad \lfloor \text{MG-REQ-F-OUT-ERR} \rfloor$$

$$\frac{\mathcal{M} \xrightarrow{a(s[p]:G)} \!\!\!\!\!\!/\ \quad}{\mathcal{M}[P] \parallel \overline{a}\langle s[p] : G\rangle \cdot H \xrightarrow{\tau}_{\mathrm{g}} \mathcal{M}[P] \parallel H} \quad \lfloor \text{MG-ACC-B-IN-ERR} \rfloor$$

$$\frac{\mathcal{M} \xrightarrow{a\langle s[p]:G\rangle} \!\!\!\!\!\!/\ \quad}{\mathcal{M}[P] \parallel \overline{a}\langle s[p] : G\rangle \cdot H \xrightarrow{\tau}_{\mathrm{g}} \mathcal{M}[P] \parallel H} \quad \lfloor \text{MG-ACC-F-IN-ERR} \rfloor$$

$$\frac{\mathcal{M} \xrightarrow{s[p_1,p_2]!l\langle v\rangle} \!\!\!\!\!\!/\ \quad}{\mathcal{M}[P \mid s[p_1]{:}\, s\langle p_1, p_2, l\langle v\rangle\rangle \cdot h] \parallel H \xrightarrow{\tau}_{\mathrm{g}} \mathcal{M}[P \mid s[p_1]{:}\, h] \parallel H} \quad \lfloor \text{MG-SEL-OUT-ERR} \rfloor$$

$$\frac{\mathcal{M} \xrightarrow{s[p_1,p_2]?l(v)} \!\!\!\!\!\!/\ \quad}{\mathcal{M}[P \mid s[p_2]{:}\, h] \parallel s\langle p_1, p_2, l\langle v\rangle\rangle \cdot H \xrightarrow{\tau}_{\mathrm{g}} \mathcal{M}[P \mid s[p_2]{:}\, h] \mid H} \quad \lfloor \text{MG-BRA-IN-ERR} \rfloor$$

Figure 7.9: The Labelled Transition System of the Monitored Network as Actions are Invalid

2. Rules ⌊MG-NEW-A⌋ and ⌊MG-JOIN⌋ say that actions $\mathsf{new}\ a : \mathtt{im}(G[p])$ and $\mathsf{join}(s[p])$ are approved/known by the local monitor, and they are the information that the monitor can inform the network.

3. Rule ⌊MG-REQ-B-OUT⌋ says when the action is a bound request approved by the monitor, the session s is newed and an invitation $\overline{a}\langle s[p] : G\rangle$ is outputted to the network.

4. Rule ⌊MG-REQ-F-OUT⌋ says when the action is a free request approved by the monitor, then the invitation $\overline{a}\langle s[p] : G\rangle$ positioning inside the local process is permitted to output to the network.

5. Rules ⌊MG-ACC-B-IN⌋ and ⌊MG-ACC-F-IN⌋ are similar. They say that when the accept action is approved by the monitor, the invitation enters the endpoint process and is in parallel with the process.

6. Rule ⌊MG-SEL-OUT⌋ says when the monitor approves the interaction message, which is inside the local queue of $s[p_1]$, this message is permitted to output to the network.

7. Rule ⌊MG-BRA-IN⌋ is dual to rule ⌊MG-SEL-OUT⌋.

8. Rule ⌊MG-PAR⌋ says that, based on Definition 5.4.4 for network parallel composition, the action happening in one network will not affect another network.

9. Rule ⌊MG-RES⌋ says if an action is valid for a network N and it results in N', and if the name of the action does not occur in the bound names of $(\nu\tilde{n})N$, then this action is also valid for $(\nu\tilde{n})N$ and results in $(\nu\tilde{n})N'$.

10. Rule ⌊MG-STR⌋ says if an action is valid for a network N_2, then, if N_1 is structural congruent to N_2, this action is also valid for N_2. Once this action fires thus makes N_2 become N_2', as the action fires in N_1, it makes N_1 become N_1', which should be structural congruent to N_2'.

For invalid actions, Figure 7.9 shows that a monitor deals with the exceptions by simply removing the invalid primitives or messages from a local process or the global transport. Here we have one important assumption particularly for ⌊MG-SEL-OUT-ERR⌋ :

> We assume there is a *timeout* mechanism for ⌊MG-SEL-OUT-ERR⌋. As a process continue producing wrong outputs up to a default *timeout*, the monitor will broadcast to all session-roles in the session with error message `session-failure`.

The *timeout* mechanism can ensure the property of *progress* of local monitored processes. It is specially for output session interaction actions, i.e. $s[p,q]!l\langle v\rangle$, but not for input actions because, as a monitor approves an input for a local process, an asynchronous local process can always take an input action to get message in even if the process itself is not ready to (or will never) take it: an asynchronous local process can always gets an input by putting it into the local buffer. However, as a local process does not take an output session interaction action obeying to the specification defined in the monitor, the output message will be taken off and the action will be stopped. If there is a corresponding receiver waiting for an input coming from this sender, it may need to wait forever if there is no mechanism of *timeout* or *message-enforcement*. The *timeout mechanism* will inform all the session participants that there is a participant failed since it produces errors up to a default time limit. Then the session will finish and will be closed. The message-enforcement mechanism is to revise the output message to the default right format obeying the specification. For example, revise the wrong receiver to the right destination, or revise the content to the right type or to satisfy the assertions. Both are not in the scope of this thesis but are topics that we should continue to work on.

Also note that, corresponding the error case of rule ⌊REQ-B-OUT⌋ defined in Figure 7.7, ⌊MG-REQ-B-OUT-ERR⌋ also defines that the process following bound request $\overline{a}(s[p] : G)$ should be cleared out.

An unmonitored network is obtained by substituting monitored processes $\mathcal{M}[P]$ with *monitor-off processes* $\mathcal{M}^\circ[P]$ where a gateway $\mathcal{M}^\circ$ is used instead of the corresponding monitor $\mathcal{M}$. The semantics for $\mathcal{M}_1^\circ[P] \xrightarrow{\ell} \mathcal{M}_2^\circ[P']$ is precisely as the one for $\mathcal{M}_1[P] \xrightarrow{\ell} \mathcal{M}_2[P']$, except that we use $\mathcal{M}_1^\circ \xrightarrow{\ell} \mathcal{M}_2^\circ$ instead of $\mathcal{M}_1 \xrightarrow{\ell} \mathcal{M}_2$.

7.3 Use Case: Monitoring OOI Instrument Commands

Since the endpoints reside in diverse administrative entities, each endpoint needs to expect the use of unsafe codes at (other) endpoints, while protocols need be executed

assuring their global correctness. Given that two interacting endpoints may be a thousand miles apart, its architectural design stipulates that this assurance is to be done through the collection of dynamic local verifications. The model is based on these principles. Each *principal* in the OOI (see Section 2.2 in Chapter 2) is modelled as a local authority executing a local process, where incoming and outgoing messages are controlled by a monitor. As depicted in Figure 7.2, monitors are *external* to endpoints, since generally they are not located under local authorities: rather monitors are part of the distributed infrastructure, protecting the integrity of global interactions which take place there.

7.3.1 Invitations of the OOI Instrument Commands

Monitors dynamically learn about which protocols to enforce during session establishment. *Local verification* of incoming/outgoing messages to/from an endpoint are done by a dedicated (external and trusted) monitor. The aim is to ensure the correctness of global interactions, while protecting the endpoint so that it sends/receives only proper messages.

Below we illustrate monitored processes of session creation and invitation (assuming shared names have already been created). $\mathcal{M}_i$ is the monitor corresponding to $Principal_i$, $i = 1, 2, 3, 4, 5$, and $\mathcal{M}_i^k$ is the status of monitor $\mathcal{M}_i$ after the kth transition from $\mathcal{M}_i^k$.

Let P_{INV} be $\overline{a_3}\langle s[Agent] : G_{IC}\rangle; \overline{a_4}\langle s[Instrument] : G_{IC}\rangle$

[step0]
$\mathcal{M}_1^0[\overline{a_2}(s[Register] : G_{IC}).\mathsf{join}(s[User]).(P_{user} \mid P_{INV})] \parallel H \parallel$
$\mathcal{M}_2^0[a_2(y_2[Register] : G_{IC}).\mathsf{join}(y_2[Register]).P_{register}] \parallel$
$\mathcal{M}_3^0[a_3(y_3[Agent] : G_{IC}).\mathsf{join}(y_3[Agent]).P_{agent}] \parallel$
$\mathcal{M}_4^0[a_4(y_4[Instrument] : G_{IC}).\overline{a_5}\langle y_4[Instrument] : G_{IC}\rangle] \parallel$
$\mathcal{M}_5^0[a_5(y_5[Instrument] : G_{IC}).\mathsf{join}(y_5[Instrument]).P_{inst}]$

[step1]
$\xrightarrow{\overline{a_2}(s[Register]:G_{IC})}_{\mathrm{g}}$
$(\nu s)(\mathcal{M}_1^1[\mathsf{join}(s[User]).P_{user} \mid P_{INV}] \parallel H \cdot \overline{a_2}\langle s[Register] : G_{IC}\rangle) \parallel$
$\mathcal{M}_2^0[a_2(y_2[Register] : G_{IC}).\mathsf{join}(y_2[Register]).P_{register}] \parallel$
$\mathcal{M}_3^0[a_3(y_3[Agent] : G_{IC}).\mathsf{join}(y_3[Agent]).P_{agent}] \parallel$
$\mathcal{M}_4^0[a_4(y_4[Instrument] : G_{IC}).\overline{a_5}\langle y_4[Instrument] : G_{IC}\rangle] \parallel$
$\mathcal{M}_5^0[a_5(y_5[Instrument] : G_{IC}).\mathsf{join}(y_5[Instrument]).P_{inst}]$

[step2]
$\xrightarrow{\mathsf{join}(s[User])a_2(s[Register]:G_{IC})}_{\mathrm{g}}$
$(\nu s)(\mathcal{M}_1^2[P_{user} \mid s[User] : \emptyset \mid P_{INV}] \parallel H \parallel$
$\mathcal{M}_2^1[\mathsf{join}(s[Register]).P_{register}\{s/y_2\}]) \parallel$
$\mathcal{M}_3^0[a_3(y_3[Agent] : G_{IC}).\mathsf{join}(y_3[Agent]).P_{agent}] \parallel$
$\mathcal{M}_4^0[a_4(y_4[Instrument] : G_{IC}).\overline{a_5}\langle y_4[Instrument] : G_{IC}\rangle] \parallel$
$\mathcal{M}_5^0[a_5(y_5[Instrument] : G_{IC}).\mathsf{join}(y_5[Instrument]).P_{inst}]$

[step3]
$\xrightarrow{\mathsf{join}(s[Register])\overline{a_3}\langle s[Agent]:G_{IC}\rangle\overline{a_4}\langle s[Instrument]:G_{IC}\rangle}_{\mathrm{g}}$
$(\nu s)(\mathcal{M}_1^3[P_{user} \mid s[User] : \emptyset] \parallel$
$H \cdot \overline{a_3}\langle s[Agent] : G_{IC}\rangle \cdot \overline{a_4}\langle s[Instrument] : G_{IC}\rangle \parallel$
$\mathcal{M}_2^2[P_{register}\{s/y_2\} \mid s[Register] : \emptyset]) \parallel$
$\mathcal{M}_3^0[a_3(y_3[Agent] : G_{IC}).\mathsf{join}(y_3[Agent]).P_{agent}] \parallel$
$\mathcal{M}_4^0[a_4(y_4[Instrument] : G_{IC}).\overline{a_5}\langle y_4[Instrument] : G_{IC}\rangle] \parallel$
$\mathcal{M}_5^0[a_5(y_5[Instrument] : G_{IC}).\mathsf{join}(y_5[Instrument]).P_{inst}]$

[step4]
$\xrightarrow{a_3(s[Agent]:G_{IC})a_4(s[Instrument]:G_{IC})}_{\mathrm{g}}$
$(\nu s)(\mathcal{M}_1^3[P_{user} \mid s[User] : \emptyset] \parallel H \parallel \mathcal{M}_2^2[P_{register}\{s/y_2\} \mid s[Register] : \emptyset] \parallel$
$\mathcal{M}_3^1[\mathsf{join}(s[Agent]).P_{agent}\{s/y_3\}] \parallel \mathcal{M}_4^1[\overline{a_5}\langle s[Instrument] : G_{IC}\rangle]) \parallel$
$\mathcal{M}_5^0[a_5(y_5[Instrument] : G_{IC}).\mathsf{join}(y_5[Instrument]).P_{inst}]$

[step5]
$\xrightarrow{\mathsf{join}(s[Agent])}_{\mathrm{g}}$
$(\nu s)(\mathcal{M}_1^3[P_{user} \mid s[User] : \emptyset] \parallel H \parallel$
$\mathcal{M}_2^2[P_{register}\{s/y_2\} \mid s[Register] : \emptyset] \parallel \mathcal{M}_3^2[P_{agent}\{s/y_3\} \mid s[Agent] : \emptyset] \parallel \mathcal{M}_4^1[\overline{a_5}\langle s[Instrument] : G_{IC}\rangle]) \parallel$
$\mathcal{M}_5^0[a_5(y_5[Instrument] : G_{IC}).\mathsf{join}(y_5[Instrument]).P_{inst}]$

[step6]
$\xrightarrow{\overline{a_5}(s[Instrument]:G_{IC})}_{\mathrm{g}}$
$(\nu s)(\mathcal{M}_1^3[P_{user} \mid s[User] : \emptyset] \parallel H \cdot \overline{a_5}\langle s[Instrument] : G_{IC}\rangle \parallel$
$\mathcal{M}_2^2[P_{register}\{s/y_2\} \mid s[Register] : \emptyset] \parallel \mathcal{M}_3^2[P_{agent}\{s/y_3\} \mid s[Register] : \emptyset]) \parallel$
$\mathcal{M}_5^0[a_5(y_5[Instrument] : G_{IC}).\mathsf{join}(y_5[Instrument]).P_{inst}] \parallel \mathcal{M}_4^2[\mathbf{0}]$

[step7]
$\xrightarrow{a_5(s[Instrument]:G_{IC})}_{\mathrm{g}}$
$(\nu s)(\mathcal{M}_1^3[P_{user} \mid s[User] : \emptyset] \parallel H \parallel \mathcal{M}_2^2[P_{register}\{s/y_2\} \mid s[Register] : \emptyset] \parallel$
$\mathcal{M}_3^2[P_{agent}\{s/y_3\} \mid s[Register] : \emptyset] \parallel \mathcal{M}_5^1[\mathsf{join}(s[Instrument]).P_{inst}\{s/y_5\}]) \parallel \mathcal{M}_4^2[\mathbf{0}]$

[step8]
$\xrightarrow{\mathsf{join}(s[Instrument])}_{\mathrm{g}}$
$(\nu s : G_{IC})(\mathcal{M}_1^3[P_{user} \mid s[User] : \emptyset] \parallel H \parallel \mathcal{M}_2^2[P_{register}\{s/y_2\} \mid s[Register] : \emptyset] \parallel$
$\mathcal{M}_3^2[P_{agent}\{s/y_3\} \mid s[Register] : \emptyset] \parallel \mathcal{M}_5^2[P_{inst}\{s/y_5\} \mid s[Instrument] : \emptyset]) \parallel \mathcal{M}_4^2[\mathbf{0}]$

In step 0, the process $\overline{a_2}(s[Register] : G_{IC}).\mathsf{join}(s[User]).(P_{user} \mid P_{INV})$ creates a fresh session s by primitive $\overline{a_2}(s[Register] : G_{IC})$, which creates session s obeying to protocol G_{IC} and, at the same time, requests endpoint a_2 to play role $Register$ in s. Then this process concurrently joins session s for playing role $User$ and continues

P_{user} and invites other endpoints to join session s through P_{INV}. Monitored process $\mathcal{M}_2^0[a_2(y_2[Register] : G_{IC}).\mathsf{join}(y_2[Register]).P_{register}]$ is ready to receive invitation for playing role $Register$ obeying to protocol G_{IC} by replacing variable y_2 with the session names. As it accepts the invitation, it joins the session as a $Register$. Monitored process $\mathcal{M}_3^0[a_3(y_3[Agent] : G_{IC}).\mathsf{join}(y_3[Agent]).P_{agent}]$ is similar to the one for playing $Register$. The monitored process

$$\mathcal{M}_4^0[a_4(y_4[Instrument] : G_{IC}).\overline{a_5}\langle y_4[Instrument] : G_{IC}\rangle]$$

does not join the session when it accepts the invitation but delegates the capability of playing role $Instrument$ to endpoint a_5. The monitored process

$$\mathcal{M}_5^0[a_5(y_5[Instrument] : G_{IC}).\mathsf{join}(y_5[Instrument]).P_{inst}]$$

is similar to those for playing roles $Register$ and $Agent$. It will accept the invitation and join the session for playing role $Instrument$.

Note that, actions in $\xrightarrow{\mathsf{join}(s[User])a_2(s[Register]:G_{IC})}_{\mathrm{g}}$ are permutable to each other, for example, it can be permuted to $\xrightarrow{a_2(s[Register]:G_{IC})\mathsf{join}(s[User])}_{\mathrm{g}}$ because they have no causal relation. Similarly, actions in $\xrightarrow{\mathsf{join}(s[Register])\overline{a_3}\langle s[Agent]:G_{IC}\rangle\overline{a_4}\langle s[Instrument]:G_{IC}\rangle}_{\mathrm{g}}$ are permutable to each other. According to the structural congruence of the network defined in Definition 5.4.6, since s is newly created, in H, there is no message belonging to session s, thus the message queue $H \cdot \overline{a_2}\langle s[Register] : G_{IC}\rangle$ can be rearranged to $\overline{a_2}\langle s[Register] : G_{IC}\rangle \cdot H$. Then invitation $\overline{a_2}\langle s[Register] : G_{IC}\rangle$ can be absorbed by the monitored process $\mathcal{M}_2^0[a_2(y_2[Register] : G_{IC}).\mathsf{join}(y_2[Register]).P_{register}]$ immediately because there is a process ready to input this invitation. The causal relation of input/output messages, which represent actions, are decided by the endpoint monitors. The transition actions in other steps are permutable for the same reason.

Also note that, after name s is created, from step 1 to step 2 the scope of name s is changed from $(\nu s)(\mathcal{M}_1^1[\mathsf{join}(s[User]).P_{user} \mid P_{INV}] \mid H \cdot \overline{a_2}\langle s[Register] : G_{IC}\rangle)$ to $(\nu s)(\mathcal{M}_1^2[P_{user} \mid s[User] : \emptyset \mid P_{INV}] \mid H \mid \mathcal{M}_2^1[\mathsf{join}(s[Register]).P_{register}\{s/y_2\}])$ since endpoint a_2 accepts the invitation coming from session s. The scope of a session ranges over those processes who currently participate in s. Note, as for monitored processes, the scope of s can be controlled by $\mathcal{M}$ through checking whether $s \in \mathcal{M}$.

In step 2, the monitored process $\mathcal{M}_1^1[\mathsf{join}(s[User]).P_{user} \mid P_{INV}]$ joins session s for playing role $User$ and becomes $\mathcal{M}_1^2[P_{user} \mid s[User] : \emptyset \mid P_{INV}]$ in which local queue specifically for session-role $s[User]$ is created. Also, the monitored process $\mathcal{M}_2^0[a_2(y_2[Register] : G_{IC}).\mathsf{join}(y_2[Register]).P_{register}]$ accepts the invitation for playing role $Register$ in session s and becomes $\mathcal{M}_2^1[\mathsf{join}(s[Register]).P_{register}\{s/y_2\}]$ in which every variable y_2 is replaced by s.

We denote $G_{IC} \upharpoonright User$ as T_{user}, $G_{IC} \upharpoonright Register$ as $T_{register}$, $G_{IC} \upharpoonright Agent$ as T_{agent}, and $G_{IC} \upharpoonright Instrument$ as T_{inst}. We show the configurations of monitors of the above

monitored processes in the follows:

[step0]

$$\begin{aligned}
\mathcal{M}_1^0 &= a_1 : \mathtt{I}(G_{IC}[User]), a_2 : \mathtt{O}(G_{IC}[Register]), a_3 : \mathtt{O}(G_{IC}[Agent]), a_4 : \mathtt{O}(G_{IC}[Instrument]) \\
\mathcal{M}_2^0 &= a_2 : \mathtt{I}(G_{IC}[Register]) \\
\mathcal{M}_3^0 &= a_3 : \mathtt{I}(G_{IC}[Agent]) \\
\mathcal{M}_4^0 &= a_4 : \mathtt{I}(G_{IC}[Instrument]), a_5 : \mathtt{O}(G_{IC}[Instrument]) \\
\mathcal{M}_5^0 &= a_5 : \mathtt{I}(G_{IC}[Instrument])
\end{aligned}$$

[step1]

$$\begin{aligned}
\mathcal{M}_1^1 &= a_1 : \mathtt{I}(G_{IC}[User]), a_2 : \mathtt{O}(G_{IC}[Register]), a_3 : \mathtt{O}(G_{IC}[Agent]), a_4 : \mathtt{O}(G_{IC}[Instrument]) \\
&\quad s[User] : T_{user}, s[Agent] : T_{agent}, s[Instrument] : T_{inst}
\end{aligned}$$

[step2]

$$\begin{aligned}
\mathcal{M}_1^2 &= a_1 : \mathtt{I}(G_{IC}[User]), a_2 : \mathtt{O}(G_{IC}[Register]), a_3 : \mathtt{O}(G_{IC}[Agent]), a_4 : \mathtt{O}(G_{IC}[Instrument]) \\
&\quad s[User]^\bullet : T_{user}, s[Agent] : T_{agent}, s[Instrument] : T_{inst} \\
\mathcal{M}_2^1 &= a_2 : \mathtt{I}(G_{IC}[Register]), s[Register] : T_{register}
\end{aligned}$$

[step3]

$$\begin{aligned}
\mathcal{M}_1^3 &= a_1 : \mathtt{I}(G_{IC}[User]), a_2 : \mathtt{O}(G_{IC}[Register]), a_3 : \mathtt{O}(G_{IC}[Agent]), a_4 : \mathtt{O}(G_{IC}[Instrument]) \\
&\quad s[User]^\bullet : T_{user} \\
\mathcal{M}_2^2 &= a_2 : \mathtt{I}(G_{IC}[Register]), s[Register]^\bullet : T_{register}
\end{aligned}$$

[step4]

$$\begin{aligned}
\mathcal{M}_3^1 &= a_3 : \mathtt{I}(G_{IC}[Agent]), s[Agent] : T_{agent} \\
\mathcal{M}_4^1 &= a_4 : \mathtt{I}(G_{IC}[Instrument]), a_5 : \mathtt{O}(G_{IC}[Instrument]), s[Instrument] : T_{inst}
\end{aligned}$$

[step5]

$$\mathcal{M}_3^2 = a_3 : \mathtt{I}(G_{IC}[Agent]), s[Agent]^\bullet : T_{agent}$$

[step6]

$$\mathcal{M}_4^2 = a_4 : \mathtt{I}(G_{IC}[Instrument]), a_5 : \mathtt{O}(G_{IC}[Instrument])$$

[step7]

$$\mathcal{M}_5^1 = a_5 : \mathtt{I}(G_{IC}[Instrument], s[Instrument] : T_{inst}$$

[step8]

$$\mathcal{M}_5^2 = a_5 : \mathtt{I}(G_{IC}[Instrument], s[Instrument]^\bullet : T_{inst}$$

As for the configurations of monitors, step 0 shows that each monitor should have *knowledge* about the endpoints who they are going to request. For example, $\mathcal{M}_1^0$ knows $a_1 : \mathtt{I}(G_{IC}[User]), a_2 : \mathtt{O}(G_{IC}[Register]), a_3 : \mathtt{O}(G_{IC}[Agent])$ and $a_4 : \mathtt{O}(G_{IC}[Instrument])$ so that she can request a_2, a_3, and a_4 for playing roles defined in protocol G_{IC}, but she can not request endpoint a_5 since she does not have the knowledge about it. Among

these monitors, only $\mathcal{M}_4^0$ with the knowledge $a_5 : \mathtt{O}(G_{IC}[Instrument])$ can request a_5 for playing role *instrument* specified under G_{IC}.

In step 1, since bound request $\overline{a_2}(s[Register] : G_{IC})$ creates a fresh name s and requests a_2 for playing role $Register$, the capabilities of playing roles $User$, $Agent$, and $Instrument$ are created and denoted by $s[User] : T_{user}, s[Agent] : T_{agent}, s[Instrument] : T_{inst}$, while the capability of playing role $Register$ is not there because it has been sent out to endpoint a_2. In step 2, the capability of playing role $User$ becomes active as $s[User]^\bullet : T_{user}$ when the process joins session s to play role $User$. And capability of $s[Register] : T_{register}$ is added to monitor M_2^1 when it accepts the invitation for playing role $Register$. In step 3, since the capabilities of $s[Agent] : T_{agent}$ and $s[Instrument] : T_{inst}$ have been sent out as requests, they are removed from the monitor. And, as explained above, the monitored process joins role $Register$ in s so that it is active as $s[Register]^\bullet : T_{register}$. Other operations of the configurations of monitors can be similarly explained.

7.3.2 Interactions of the OOI Instrument Commands

It follows an illustration of some actions of the session instrument command. For simplicity, we use P_{role}^k to represent the status after the kth action of process P_{role}.

[step0]
$(\nu s)(H \parallel \mathcal{M}_1^3[P_{user} \mid s[User] : \emptyset] \parallel \mathcal{M}_2^2[P_{register} \mid s[Register] : \emptyset] \parallel$
$\mathcal{M}_3^2[P_{agent} \mid s[Agent] : \emptyset] \parallel \mathcal{M}_5^2[P_{inst} \mid s[Instrument] : \emptyset])$

[step1]
$\xrightarrow{\tau}_{\mathrm{g}}$
$(\nu s)(H \parallel \mathcal{M}_1^3[P_{user}^1 \mid s[User] : s\langle User, Register, \langle v_{int}\rangle\rangle] \parallel$
$\mathcal{M}_2^2[P_{register} \mid s[Register] : \emptyset] \parallel \mathcal{M}_3^2[P_{agent} \mid s[Agent] : \emptyset] \parallel \mathcal{M}_5^2[P_{inst} \mid s[Instrument] : \emptyset])$

[step2]
$\xrightarrow{s[User,Register]!\langle v_{int}\rangle}_{\mathrm{g}}$
$(\nu s)(H \cdot s\langle User, Register, \langle v_{int}\rangle\rangle \parallel \mathcal{M}_1^4[P_{user}^1 \mid s[User] : \emptyset] \parallel$
$\mathcal{M}_2^2[P_{register} \mid s[Register] : \emptyset] \parallel \mathcal{M}_3^2[P_{agent} \mid s[Agent] : \emptyset] \parallel \mathcal{M}_5^2[P_{inst} \mid s[Instrument] : \emptyset])$

[step3]
$\xrightarrow{s[User,Register]?(v_{int})}_{\mathrm{g}}$
$(\nu s)(H \parallel \mathcal{M}_1^4[P_{user}^1 \mid s[User] : \emptyset] \parallel \mathcal{M}_2^3[P_{register} \mid s[Register] : s\langle User, Register, \langle v_{int}\rangle\rangle] \parallel$
$\mathcal{M}_3^2[P_{agent} \mid s[Agent] : \emptyset] \parallel \mathcal{M}_5^2[P_{inst} \mid s[Instrument] : \emptyset])$

[step4]
$\xrightarrow{\tau}_{\mathrm{g}}$
$(\nu s)(H \parallel \mathcal{M}_1^4[P_{user}^1 \mid s[User] : \emptyset] \parallel$
$\mathcal{M}_2^3[P_{register}^1 \mid s[Register] : \emptyset] \parallel \mathcal{M}_3^2[P_{agent} \mid s[Agent] : \emptyset] \parallel \mathcal{M}_5^2[P_{inst} \mid s[Instrument] : \emptyset])$

[step5]
$\xrightarrow{\tau}_{\mathrm{g}}$
$(\nu s)(H \parallel \mathcal{M}_1^4[P_{user}^1 \mid s[User] : \emptyset] \parallel$
$\mathcal{M}_2^3[P_{register}^2 \mid s[Register] : s\langle Register, User, \langle 10\rangle\rangle] \parallel$
$\mathcal{M}_3^2[P_{agent} \mid s[Agent] : \emptyset] \parallel \mathcal{M}_5^2[P_{inst} \mid s[Instrument] : \emptyset])$

[step6]
$\xrightarrow{s[Register,User]!\langle 10\rangle}_{\mathrm{g}}$
$(\nu s)(H \cdot s\langle Register, User, \langle 10\rangle\rangle \parallel$
$\mathcal{M}_1^4[P_{user}^1 \mid s[User] : \emptyset] \parallel \mathcal{M}_2^4[P_{register}^2 \mid s[Register] : \emptyset] \parallel \mathcal{M}_3^2[P_{agent} \mid s[Agent] : \emptyset] \parallel$
$\mathcal{M}_5^2[P_{inst} \mid s[Instrument] : \emptyset])$

[step7]
$\xrightarrow{s[Register,User]?(10)}_{\mathrm{g}}$
$(\nu s)(H \parallel \mathcal{M}_1^5[P_{user}^1 \mid s[User] : s\langle Register, User, \langle 10\rangle\rangle] \parallel$
$\mathcal{M}_2^4[P_{register}^2 \mid s[Register] : \emptyset] \parallel$
$\mathcal{M}_3^2[P_{agent} \mid s[Agent] : \emptyset] \parallel \mathcal{M}_5^2[P_{inst} \mid s[Instrument] : \emptyset])$

[step8]
$\xrightarrow{\tau}_{\mathrm{g}}$
$(\nu s)(H \parallel \mathcal{M}_1^5[P_{user}^2 \mid s[User] : \emptyset] \parallel$
$\mathcal{M}_2^4[P_{register}^2 \mid s[Register] : \emptyset] \parallel$
$\mathcal{M}_3^2[P_{agent} \mid s[Agent] : \emptyset] \parallel \mathcal{M}_5^2[P_{inst} \mid s[Instrument] : \emptyset])$

In step 0, since it continues with the operations of invitation, $\mathcal{M}_4^2[\mathbf{0}] \equiv \mathbf{0}$ (based on Definition 7.2.5), $\mathcal{M}_4^2$ is neglected. In step 1, P_{user} *internally* sends an output message $s\langle User, Register, \langle v_{int}\rangle\rangle$ and puts it in its local queue as

$$s[User] : s\langle User, Register, \langle v_{int}\rangle\rangle.$$

Since this internal output action is invisible globally, it is a silent action, i.e. τ action, of the LTS of the monitored network. Then in step 2, the message is sent out from

the monitored process, thus the visible action $s[User, Register]!\langle v_{int}\rangle$ happens, and the local queue becomes empty as $s[User] : \emptyset$, and the message is put in the global queue as $H \cdot s\langle User, Register, \langle v_{int}\rangle\rangle$. In step 3, since visible action $s[User, Register]?(v_{int})$ happens, the messasge $s\langle User, Register, \langle v_{int}\rangle\rangle$ is removed from the global queue, which becomes H, and enters the local process $P_{register}$, thus the monitored process of playing role *Register* becomes $\mathcal{M}_2^3[P_{register} \mid s[Register] : s\langle User, Register, \langle v_{int}\rangle\rangle]$. In step 4, message $s\langle User, Register, \langle v_{int}\rangle\rangle$ is absorbed by process $P_{register}$. Thus $P_{register}$ becomes $P^1_{register}$ and its local queue becomes empty as $s[Register] : \emptyset$. Other operations of the following steps are similar.

Up to now, $User$ and *Register* finish their first interaction: $User$ sends v_{int} to *Register*, then *Register* returns 10 back to $User$. Other value-passing and branching interactions afterwards are similar. Recall that we denote $G_{IC} \upharpoonright User$ as T_{user}, $G_{IC} \upharpoonright Register$ as $T_{register}$, $G_{IC} \upharpoonright Agent$ as T_{agent}, and $G_{IC} \upharpoonright Instrument$ as T_{inst}. For convenience we let T^k_{role} to represent the kth action of endpoint assertion T_{role}.

[step0]

$$\begin{aligned}\mathcal{M}_1^3 &= a_1 : \mathtt{I}(G_{IC}[User]), a_2 : \mathtt{O}(G_{IC}[Register]), a_3 : \mathtt{O}(G_{IC}[Agent]), a_4 : \mathtt{O}(G_{IC}[Instrument]),\\ &\quad s[User]^\bullet : T_{user} = Register!(x_{int} : \mathsf{InterfaceId}).T^1_{user}\\ \mathcal{M}_2^2 &= a_2 : \mathtt{I}(G_{IC}[Register]),\\ &\quad s[Register]^\bullet : T_{register} = User?(x_{int} : \mathsf{InterfaceId}).T^1_{register}\end{aligned}$$

[step1]

$$\begin{aligned}\mathcal{M}_1^4 &= a_1 : \mathtt{I}(G_{IC}[User]), a_2 : \mathtt{O}(G_{IC}[Register]), a_3 : \mathtt{O}(G_{IC}[Agent]), a_4 : \mathtt{O}(G_{IC}[Instrument]),\\ &\quad s[User]^\bullet : T^1_{user}\{v_{int}/x_{int}\} = Register?(x_n : \mathsf{Int})\langle\!\langle x_n > 0\rangle\!\rangle.T^2_{user}\{v_{int}/x_{int}\}\end{aligned}$$

[step2]

$$\begin{aligned}\mathcal{M}_2^3 &= a_2 : \mathtt{I}(G_{IC}[Register]),\\ &\quad s[Register]^\bullet : T^1_{register}\{v_{int}/x_{int}\} = User!(x_n : \mathsf{Int})\langle\!\langle x_n > 0\rangle\!\rangle.T^2_{register}\{v_{int}/x_{int}\}\end{aligned}$$

[step3]

$$\begin{aligned}\mathcal{M}_2^4 &= a_2 : \mathtt{I}(G_{IC}[Register]),\\ &\quad s[Register]^\bullet : T^2_{register}\{v_{int}/x_{int}\}\{10/x_n\} = \mathsf{end}\{v_{int}/x_{int}\}\{10/x_n\} = \mathsf{end}\end{aligned}$$

[step4]

$$\begin{aligned}\mathcal{M}_1^5 &= a_1 : \mathtt{I}(G_{IC}[User]), a_2 : \mathtt{O}(G_{IC}[Register]), a_3 : \mathtt{O}(G_{IC}[Agent]), a_4 : \mathtt{O}(G_{IC}[Instrument]),\\ &\quad s[User]^\bullet : T^2_{user}\{v_{int}/x_{int}\}\{10/x_n\}\end{aligned}$$

In step 0, monitor $\mathcal{M}_1^3$ has active capability $s[User]^\bullet$, which is able to send a message to role *Register* with content of type $\mathsf{InterfaceId}$, then local specification T^1_{user} is ready for

the next action. Similarly, monitor $\mathcal{M}_2^2$ has active capability $s[Register]^\bullet$, which is able to receive message of type of InterfaceId from role $User$, then local specification $T^1_{register}$ is ready for the next action. Step 1 describes when action $s[User, Register]!\langle v_{int}\rangle$ takes place, then $Register!(x_{int} : \mathsf{InterfaceId})$ is removed, and T^1_{user} becomes $T^1_{user}\{v_{int}/x_{int}\}$, which becomes $Register?(x_n : \mathsf{Int})\langle\langle x_n > 0\rangle\rangle.T^2_{user}\{v_{int}/x_{int}\}$ for the next input action from role $Register$. Other operations of configurations of monitors above are similar and they all obey the LTS of the monitors, defined in Figure 7.4.

7.4 Safety, Transparency, and Session Fidelity

This section states and proves the key theorems, including the receivability and coherence of the network, properties of local and global safety, local and global transparency, and session fidelity. They are the properties that our monitoring mechanism can enforce.

7.4.1 Local Safety and Transparency

We first list the properties that monitors guarantee for local configurations. Hereafter, we write $\mathcal{L}$, a located process, for either a monitored process $\mathcal{M}[P]$ or a monitor-off process $\mathcal{M}^\circ[P]$.

Definition 7.4.1. Define $\mathcal{L} \xrightarrow{\ell}$ as $\mathcal{L}$ approves the action ℓ and it implies $\exists\mathcal{L}'$ such that $\mathcal{L} \xrightarrow{\ell} \mathcal{L}'$. Similarly, define $\mathcal{M} \xrightarrow{\ell}$ as $\mathcal{M}$ approves the action ℓ and it implies $\exists\mathcal{M}'$ such that $\mathcal{M} \xrightarrow{\ell} \mathcal{M}'$.

Definition 7.4.2 (conformance). A relation $\mathcal{R}$ between a located process $\mathcal{L}$ and its monitor $\mathcal{M}$ is a *conformance* if whenever $\mathcal{L}\ \mathcal{R}\ \mathcal{M}$, we have

1. Given an output ℓ, if $\mathcal{L} \xrightarrow{\ell}$ then $\mathcal{M} \xrightarrow{\ell}$.
2. Given an input ℓ, if $\mathcal{M} \xrightarrow{\ell}$ then $\mathcal{L} \xrightarrow{\ell}$.
3. For any ℓ, if $\mathcal{L} \xrightarrow{\ell} \mathcal{L}'$ and $\mathcal{M} \xrightarrow{\ell} \mathcal{M}'$ then $\mathcal{L}'\ \mathcal{R}\ \mathcal{M}'$.

If $\mathcal{M}\ \mathcal{R}\ \mathcal{L}$ for a conformance $\mathcal{R}$, we write $\mathcal{M} \models \mathcal{L}$, and say $\mathcal{L}$ *conforms to* $\mathcal{M}$.

Conformance says that a located process is a process that only sends "good" messages and can at least receive "good" messages (which have been approved by its own monitor). The following theorem says every monitored process conforms to the specification given by its own monitor.

Theorem 7.4.3 (local safety (local conformance)). $\mathcal{M} \models \mathcal{M}[P]$ *for all* $\mathcal{M}$ *and* P.

Proof. The proof and auxiliary theorems are in Appendix B.1.

Definition 7.4.4. Let $\mathcal{R}$ be a relation between a monitor and a pair of located processes. Then $\mathcal{R}$ is a *monitored strong bisimulation* if the following holds for each $\mathcal{M}$, $\mathcal{L}_1$ and $\mathcal{L}_2$ related by $\mathcal{R}$. Below we assume $i \neq j$, with $i, j \in \{1, 2\}$.

1. If $\mathcal{L}_i \xrightarrow{\ell} \mathcal{L}_i'$ s.t. ℓ is an output/τ then $\mathcal{M} \xrightarrow{\ell} \mathcal{M}'$, $\mathcal{L}_j \xrightarrow{\ell} \mathcal{L}_j'$ and $(\mathcal{M}', \mathcal{L}_i', \mathcal{L}_j') \in \mathcal{R}$.
2. If $\mathcal{L}_i \xrightarrow{\ell} \mathcal{L}_i'$, $\mathcal{M} \xrightarrow{\ell} \mathcal{M}'$ s.t. ℓ is an input then $\mathcal{L}_j \xrightarrow{\ell} \mathcal{L}_j'$ and. $(\mathcal{M}', \mathcal{L}_i', \mathcal{L}_j') \in \mathcal{R}$.

If $\mathcal{M}$, $\mathcal{L}_1$ and $\mathcal{L}_2$ are in a monitored strong bisimulation then $\mathcal{L}_1$ *and* $\mathcal{L}_2$ *are bisimilar under* $\mathcal{M}$, denoted $\mathcal{M} \models \mathcal{L}_1 \sim \mathcal{L}_2$.

The following states that a monitored-process and a monitor-off process behave precisely in the same way if the latter already conforms to the monitor's specification.

Theorem 7.4.5 (local transparency). *If* $\mathcal{M} \models \mathcal{M}^\circ[P]$, *then* $\mathcal{M} \models \mathcal{M}^\circ[P] \sim \mathcal{M}[P]$.

Proof. The proof and auxiliary theorems are in Appendix B.1.

Note that a process P that can be validated by $\Gamma \vdash P \triangleright \Delta$ (in the sense of (22)) is guaranteed to behave correctly and thus satisfies $\mathcal{M} \models \mathcal{M}^\circ[P]$ for $\mathcal{M} = \Gamma, \Delta$.

7.4.2 Global Safety, Transparency and Session Fidelity

This section shows that global message flows inside the monitored network are safe in spite of untrusted endpoints. To make exact the notion of global correctness, $\xrightarrow{\ell}_{\mathrm{g}}$ is used for describing $N \xrightarrow{\ell}_{\mathrm{g}} N'$ by following the transitions for the monitored network in Figure 7.8. Note that both N and N' are the monitored network.

Definition 7.4.6 (receivable network). We say N is receivable if there exist a sequence of actions $\ell_1 \dots \ell_k$ consisting of all the inputs corresponding to the pending messages in N and joins, such that $N \xrightarrow{\ell_1 \dots \ell_k}_{\mathrm{g}} N'$.

Lemma 7.4.7 (receivability). Let N be a monitored network and ℓ be an input action. If N is receivable and $N \xrightarrow{\ell}_{\mathrm{g}} N'$, then N' is receivable.

Proof. It is proved in Appendix B.2.

Definition 7.4.8. Define $s[p]^\circ$ stand for either $s[p]$ or $s[p]^\bullet$.

Definition 7.4.9 ($T' \subseteq T$ w.r.t a session-role). We say $T' \subseteq T$ w.r.t $s[p]$ if, for some $\mathcal{M}$ containing $s[p]$, we have either (I) $s[p]^\circ : T \xrightarrow{\ell_1 ... \ell_n} s[p]^\circ : T'$ for some sequence of actions $\ell_1 ... \ell_n$ or (II) $T \curvearrowright T'$.

Note that, when T, T' are clearly for the same session-role, we simply write $T' \subseteq T$.

Here we define a group of monitors which are parts of the network as follows:

Definition 7.4.10 (a group of monitors). Define a group of monitors as

$$\mathbb{M} = \prod_{i \in I} \mathcal{M}_i = \mathcal{M}_1 \parallel \ldots \parallel \mathcal{M}_k, I = \{1, \ldots, k\}$$

the monitors existing in parallel in the network.

Remark 7.4.11. Since every monitor contains specifications Γ, Δ, $\mathbb{M}$ can also be viewed as a set of specifications.

Definition 7.4.12 (consistency of a group of monitors). Let $\mathbb{M}$ be a group of monitors. Let $\forall \mathcal{M}_i \in \mathbb{M}$, $\mathcal{M}_i = \Gamma_i, \Delta_i$. If all of the following conditions hold,

1. (a) $\forall \mathcal{M}_i \in \mathbb{M}$, if $a : \mathtt{im}(G[p]) \in \mathcal{M}_i$, then there is no other $a : \mathtt{I}(G[p]) \in \mathcal{M}_j$ or $a : \mathtt{IO}(G[p]) \in \mathcal{M}_j$, $\mathcal{M}_j \in \mathbb{M}$.
   (b) $\forall \mathcal{M}_i \in \mathbb{M}$, if $\forall s, s[p]^\circ : T \in \mathcal{M}_i$, then $s[p]^\circ \notin \mathcal{M}_j$, $j \neq i$, $\mathcal{M}_j \in \mathbb{M}$.
2. If $a : \mathtt{O}(G[p]) \in \mathcal{M}_i$, then there exists $\mathcal{M}_j$ such that $a : \mathtt{im}(G[p]) \in \mathcal{M}_j$.
3. Assume $p_1 \neq p_2$. As $\Delta_i(s[p_1]^\circ) = T$ and
$$p_2!\{l_k(x_k : S_k)\langle\langle A_k \rangle\rangle.G_k \upharpoonright p_2\}_{k \in I} \subseteq T,$$
there exists j such that $\Delta_j(s[p_2]^\circ) = T'$ and
$$p_1?\{l_k(x_k : S_k)\langle\langle A'_k \rangle\rangle.G_k \upharpoonright p_1\}_{k \in I} \subseteq T'$$
with $A_k\{v/x_k\} \downarrow \mathsf{true}$ iff $A'_k\{v/x_k\} \downarrow \mathsf{true}$ for some v.
4. $\bigcup_i \Delta_i = \bigcup_{1 \leq j \leq n} \{s_j[p_{j1}] : T_{j1}, .., s_j[p_{jk}] : T_{jk}, .., s_j[p_{jm_j}] : T_{jm_j}\}$ where, for each s_j, there is G_j such that $G_j \upharpoonright p_{jk} = T_{jk}$ for $1 \leq k \leq m_j$.

then the group of monitors, $\mathbb{M}$, is consistent.

Note that for rule 3, we only ask that, whenever there is an output, there is a corresponding input because the monitors should at least ensure that the outputs from local processes will not output unexpected messages to the network. By considering the following global specification,

$$G = p_1 \to p_2 : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.p_2 \to p_3 : \{l'_j(x'_j : S'_j)\langle\langle B_j\rangle\rangle.G'_{ij}\}_{j\in I}\}_{i\in I}$$

assume s follows G. After applying projection rules defined in Definition 6.5.1, we have

$$\begin{array}{lcl} s[p_1] & : & p_2!\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G'_{ij} \upharpoonright p_1\}_{i\in I} \\ s[p_2] & : & p_1?\{l_i(x_i : S_i)\langle\langle A'_i\rangle\rangle.p_3!\{l'_j(x'_j : S'_j)\langle\langle B_j\rangle\rangle.G'_{ij} \upharpoonright p_2\}_{j\in I}\}_{i\in I} \\ s[p_3] & : & p_2?\{l'_j(x'_j : S'_j)\langle\langle B'_j\rangle\rangle.G'_{ij} \upharpoonright p_3\}_{j\in I} \end{array}$$

where sending to $p_3!$ at $s[p_2]$ is not permutable to receiving from $p_1?$. Since the consistency of monitors requires that an output should correspond to an input, rule 3 requires that, when a sending action is in a monitor's specification, a corresponding receiving action is in some monitor's specification. Moreover, when a sending action is ready to send a message, there exists a corresponding receiving action at some monitor to input this message.

Definition 7.4.13 (monitored network coherence). N is *coherent* if all of its pending messages are *receivable* up to *permutation* of actions $\curvearrowright$ and, after these messages have been received, the resulting group of monitors which guard all local processes in N, say $\mathbb{M}$, is consistent.

The *coherence* above states that, after all messages are cleared out of the network, the monitors of the network satisfy the following rules: rule 1: An input mode shared name with particular specification is unique to the network (see Lemma 7.4.15), and similarly, a session-role is unique to the network such that no two different locations share a common role in a common session; rule 2: As long as an `O`-mode shared name exists, its dual mode (i.e. `I`/`IO`-mode) shared name co-exists; rule 3: Each output action in a specification has its corresponding input action in some other specification; rule 4: For each session name, the local specifications for the roles of its participants is the result of projecting a common global specification on their roles.

Convention 7.4.14. We say an element is unique to the network N whenever $\mathcal{M}_i \in N$, $\mathcal{M}_i$ has one and only one of this element, and $\forall \mathcal{M}_j \in N$, $j \neq i$, $\mathcal{M}_j$ does not have this element.

Compared to Definition 5.4.2 of composability, the composable network, say $N_1 \parallel N_2$ may not be coherent when they are composed in parallel. The rule 1 of Definition 7.4.12 implies that the $a : \mathtt{im}(G[p])$ is unique to the network, which is proved by the following Lemma:

Lemma 7.4.15 (global uniqueness of names). Assume every local domain has a network-wise unique identity, which will be attached to a local name when the name is created. If $a : \mathtt{im}(G[p])$ is unique to a local monitor guarding a local domain, which means there is no more $a : \mathtt{I}(G[p])$ or $a : \mathtt{IO}(G[p])$ in that monitor, then it is unique to the network. Similarly, if s is unique to a local monitor, it is unique to the network.

Proof. As Figure 7.1 shows, there is a routing mechanism in the network. Distributed routers are responsible for both routing and registering individual components in different domains. As long as a local monitor approves name n, which is either a shared name or a session name, it is fresh to the local domain, say A. When this name is announced to the network (with rules ⌊MG-NEW-A⌋ and ⌊MG-REQ-B-OUT⌋ in Figure 7.8), theoretically we can assume the identity of the domain is attached to this name, e.g. $n.A$, where A is unique in the network. Therefore, this name (i.e. $n.A$) is unique globally. With the same reasoning above, s is unique to the network as long as it is unique to a local monitor.

Since we want to abstract the actions as simple as possible, we neglect the details of routing mechanism in this thesis.

The property of global safety says that, as, at the very beginning, the network is protected by consistent monitors, the coherence of the monitored network is preserved by the interactions that can happen in a monitored network. In brief, it states that a fully monitored network behaves well with respect to the given global specification.

Theorem 7.4.16 (global safety). Let N be coherent. Then $N \xrightarrow{\ell}_{\mathrm{g}} N'$ implies N' is coherent.

Proof. It is proved in Appendix B.2.

N is *locally conformant* if, for each monitored process $\mathcal{M}_i[P_i]$ in N, we have $\mathcal{M}_i \models \mathsf{erase}(\mathcal{M}_i)[P_i]$. If N is coherent and locally conformant and such that $N \xrightarrow{\ell_1}_{\mathrm{g}} \cdots \xrightarrow{\ell_n}_{\mathrm{g}} N'$ then N' is also coherent and locally conformant. This property is formally defined and proved in Appendix B.3. Let $\sim$ be the standard strong bisimilarity defined by $\xrightarrow{\ell}_{\mathrm{g}}$. Then we have:

Theorem 7.4.17 (global transparency). Suppose N is coherent and locally conformant. Then $N \sim \mathsf{erase}(N)$.

Proof. It is proved in Appendix B.3.

The property of global transparency states that a monitored network and an unmonitored network have equivalent behaviour when the latter is well-behaved with respect to the same (collection of) specifications. When a new session is generated, it is associated with a global protocol G. In a monitored network, we expect all message flows for this session always follow G. This notion is called *session fidelity* (86).

Note as Remark 5.4.7 says, the viewpoints of local and global are dual. Define the processes that guarded by $\mathbb{M}$ as:

Definition 7.4.18. Write $\mathsf{P}(\mathcal{M})$ for the set of processes governed by monitor $\mathcal{M}$, and write $\mathsf{P}(\mathbb{M})$ to denote the set of processes governed by the group of monitors $\mathbb{M}$.

Definition 7.4.19 (labelled transition of $\mathbb{M}$). $\mathbb{M} \xrightarrow{\ell} \mathbb{M}'$ if and only if there exists $\mathcal{M} \in \mathbb{M}$ s.t. $\mathcal{M} \xrightarrow{\ell} \mathcal{M}'$ and $\mathcal{M}' \in \mathbb{M}'$.

Before introducing the theorem of sesssion fidelity, to simplify and clarify the relationships between local monitored processes and the global transport, *configuration* is introduced. It is the pair of $\mathbb{M}$ and H, to illustrate the idea of session fidelity.

Definition 7.4.20 (configuration). A configuration is denoted by $\Phi = \mathbb{M}; H$, in which the group of monitors correspond to H. In other words, all messages corresponding to the actions guarded by $\mathbb{M}$ are in H.

A Φ thus guides and captures the behaviours in the network.

Definition 7.4.21. For any $\mathbb{M}_1$ and $\mathbb{M}_2$, we write $\mathbb{M}_1, \mathbb{M}_2$ as a standard union of $\mathbb{M}_1$ and $\mathbb{M}_2$.

Lemma 7.4.22. Assume $\mathbb{M}_1$ corresponds to H_1 and $\mathbb{M}_2$ corresponds to H_2. As long as $\mathsf{P}(\mathbb{M}_1) \cap \mathsf{P}(\mathbb{M}_2) = \emptyset$, there is no common message existing in H_1 and H_2.

Proof. Based on Definition 7.4.20, for $i \in \{1, 2\}$, if the interactions, including outputs and their corresponding inputs, are approved by one $\mathbb{M}_i$, all messages corresponding to these actions are in one H_i. For $i, j \in \{1, 2\}, i \neq j$, as an output action is approved by $\mathbb{M}_i$ so that its corresponding message is put into H_i, and its input, the duality of the output, if it is approved by $\mathbb{M}_j$, then this message will leave H_i and enter H_j and will be absorbed by some monitored process guarded by a monitor in $\mathbb{M}_j$. Thus overall there is no common message existing in H_1 and H_2. ■

With the definitions and lemma above, we define the parallel composition of configurations:

Definition 7.4.23 (parallel composition of configurations). Assume $\Phi_1 = \mathbb{M}_1 \; ; \; H_1$ and $\Phi_2 = \mathbb{M}_2 \; ; \; H_2$ are given, we say Φ_1 and Φ_2 are *composable* whenever $\mathsf{P}(\mathbb{M}_1) \cap \mathsf{P}(\mathbb{M}_2) = \emptyset$. If Φ_1 and Φ_2 are composable, define the composition of Φ_1 and Φ_2 as: $\Phi_1 \odot \Phi_2 = \mathbb{M}_1, \mathbb{M}_2 \; ; \; H_1 \cdot H_2$.

$H_1 \cdot H_2$ is defined in Definition 5.4.3. The behaviour of each endpoint in a network is guided by $\mathbb{M}$ (specifications), and is observed by H (global transport). We define the LTS of configurations in Figure 7.10. All rules are straightforward from the LTS of monitors (Figure 7.4, Chapter 7) and the one of the dynamic network (Figure 5.7, Chapter 5). The global observable transition $\xrightarrow{\ell}_{\mathrm{g}}$ for some action ℓ has been introduced in the LTS for the monitored network, defined in Figure 7.8.

- Rule New informs the global transport that a fresh name a is created.
- Rules [Req-b/f] indicates that when some endpoint monitor approves a bound or free request to go to the network, the configuration inputs this request as an invitation to the global queue. Note that [Req-b] also creates and hides a fresh session s.
- Rules [Acc-b/f] indicates that, when the invitation has been accepted by an endpoint in the network, the invitation is taken off from the global queue (externally). Note that, we can only observe the invitation leaves the global queue but do not know who accepts it: only $\mathbb{M}$ can tell which endpoint accepts this invitation.
- The concepts of rules [Sel, Bra] are similar to the concepts of rules [Req-b/f, Acc-b/f].
- The concepts of rules [SelN, BraN] are similar to rule [Sel, Bra], except that they are carrying names.
- Rule Par says if Φ_1 and Φ_3 are composable (see Definition 7.4.23), after Φ_1 becomes as Φ_2, they are still composable.

Definition 7.4.24 (configurational consistency). A configuration $\Phi = \mathbb{M}; H$ is *configurationally consistent* whenever

1. H is empty and $\mathbb{M}$ is consistent, or

$$[\text{New}]\quad \frac{\mathtt{im} \in \{\mathtt{I}, \mathtt{IO}\},\ \mathbb{M} \xrightarrow{\mathsf{new}\ a{:}\mathtt{im}(G[p])} \mathbb{M}'}{\mathbb{M}\,;H \xrightarrow{\mathsf{new}\ a{:}\mathtt{im}(G[p])}_{\mathrm{g}} \mathbb{M}'\,;(\nu a)(H)}$$

$$[\text{Req-b}]\quad \frac{\mathbb{M} \xrightarrow{\overline{a}(s[p]:G)} \mathbb{M}'}{\mathbb{M}\,;H \xrightarrow{\overline{a}(s[p]:G)}_{\mathrm{g}} \mathbb{M}'\,;H\cdot\overline{a}\langle s[p]:G\rangle}$$

$$[\text{Req-f}]\quad \frac{\mathbb{M} \xrightarrow{\overline{a}\langle s[p]:G\rangle} \mathbb{M}'}{\mathbb{M}\,;H \xrightarrow{\overline{a}\langle s[p]:G\rangle}_{\mathrm{g}} \mathbb{M}'\,;H\cdot\overline{a}\langle s[p]:G\rangle}$$

$$[\text{Acc-b}]\quad \frac{\mathbb{M} \xrightarrow{a(s[p]:G)} \mathbb{M}'}{\mathbb{M}\,;\overline{a}\langle s[p]:G\rangle\cdot H \xrightarrow{a(s[p]:G)}_{\mathrm{g}} \mathbb{M}'\,;H}$$

$$[\text{Acc-f}]\quad \frac{\mathbb{M} \xrightarrow{a\langle s[p]:G\rangle} \mathbb{M}'}{\mathbb{M}\,;\overline{a}\langle s[p]:G\rangle\cdot H \xrightarrow{a\langle s[p]:G\rangle}_{\mathrm{g}} \mathbb{M}'\,;H}$$

$$[\text{Sel}]\quad \frac{\mathbb{M} \xrightarrow{s[p_1,p_2]!l\langle v\rangle} \mathbb{M}'}{\mathbb{M}\,;H \xrightarrow{s[p_1,p_2]!l\langle v\rangle}_{\mathrm{g}} \mathbb{M}'\,;H\cdot s\langle p_1,p_2,l\langle v\rangle\rangle}$$

$$[\text{SelN}]\quad \frac{\mathbb{M} \xrightarrow{s[p_1,p_2]!l\langle a\rangle} \mathbb{M}'}{\mathbb{M}\,;H \xrightarrow{s[p_1,p_2]!l\langle a\rangle}_{\mathrm{g}} \mathbb{M}'\,;H\cdot s\langle p_1,p_2,l\langle a\rangle\rangle}$$

$$[\text{Bra}]\quad \frac{\mathbb{M} \xrightarrow{s[p_1,p_2]?l(v)} \mathbb{M}'}{\mathbb{M}\,;s\langle p_1,p_2,l\langle v\rangle\rangle\cdot H \xrightarrow{s[p_1,p_2]?l(v)}_{\mathrm{g}} \mathbb{M}'\,;H}$$

$$[\text{BraN}]\quad \frac{\mathbb{M} \xrightarrow{s[p_1,p_2]?l(a)} \mathbb{M}'}{\mathbb{M}\,;s\langle p_1,p_2,l\langle a\rangle\rangle\cdot H \xrightarrow{s[p_1,p_2]?l(a)}_{\mathrm{g}} \mathbb{M}'\,;H}$$

$$[\text{Par}]\quad \frac{\Phi_1 \xrightarrow{\ell}_{\mathrm{g}} \Phi_2}{\Phi_1 \odot \Phi_3 \xrightarrow{\ell}_{\mathrm{g}} \Phi_2 \odot \Phi_3} \qquad [\text{Tau}]\quad \frac{\mathbb{M} \xrightarrow{\tau} \mathbb{M}}{\mathbb{M};H \xrightarrow{\tau}_{\mathrm{g}} \mathbb{M};H}$$

Figure 7.10: The Labelled Transition System of Configurations

2. H is not empty, the sequence of messages in H are receivable to $\mathbb{M}$, and after receiving all messages in H with $\mathbb{M} \xrightarrow{\ell_1..\ell_n} \mathbb{M}'$, where $\ell_i, i = \{1, ..., n\}$ are inputs and, $\forall m \in H$, $\exists \ell \in \ell_1 \cdot ... \cdot \ell_n$ such that ℓ corresponds to m, we have $\mathbb{M}'$ is consistent.

In other words, $\mathbb{M}; H$ is configurationally consistent if, in each of its derivatives, all messages in the transport can be "received" by some monitors in $\mathbb{M}$ and, after absorbing all these messages, the resulting $\mathbb{M}'$ is still consistent.

As for the satisfaction relation of the partial network $(\mathsf{N}_s, \mathsf{N}_s{}'...)$ [1], if N_s is a partial network, $\models \mathsf{N}_s : \mathbb{M}$ means that monitors in $\mathbb{M}$ allow all outputs from the local processes in N_s, and all the local processes in N_s are *ready* to receive the inputs which monitors in $\mathbb{M}$ indicate. The output and intput satisfactions described above are preserved by transition.

Definition 7.4.25 (partial satisfaction). A relation $\mathcal{R}$ from partial networks to specifications of monitors in $\mathbb{M}$ is a *satisfaction* when $\mathsf{N}_s \mathcal{R}\ \mathbb{M}$ implies:

1. If $\mathbb{M} \xrightarrow{\ell} \mathbb{M}'$ for an input ℓ and $\exists P_j \in \mathsf{N}_s$ has an input at $\mathsf{sbj}(\ell)$, then $P_j \xrightarrow{\ell} P_j'$, which makes $\mathsf{N}_s = P_j \parallel \prod_{i \neq j} P_i$ becomes $\mathsf{N}_s{}' = P_j' \parallel \prod_{i \neq j} P_i$ such that $\mathsf{N}_s{}' \mathcal{R}\ \mathbb{M}'$.
2. If $\exists P_j \in \mathsf{N}_s$ and $P_j \xrightarrow{\ell} P_j'$ for an output ℓ, which makes $\mathsf{N}_s = P_j \parallel \prod_{i \neq j} P_i$ becomes $\mathsf{N}_s{}' = P_j' \parallel \prod_{i \neq j} P_i$, then $\mathbb{M} \xrightarrow{\ell} \mathbb{M}'$ such that $\mathsf{N}_s{}' \mathcal{R}\ \mathbb{M}'$.
3. If $\exists P_j \in \mathsf{N}_s$ and $P_j \xrightarrow{\tau} P_j'$, which makes $\mathsf{N}_s = P_j \parallel \prod_{i \neq j} P_i$ becomes $\mathsf{N}_s{}' = P_j' \parallel \prod_{i \neq j} P_i$, then $\mathbb{M} \xrightarrow{\tau} \mathbb{M}'$ such that $\mathsf{N}_s{}' \mathcal{R}\ \mathbb{M}'$ (i.e. $\mathsf{N}_s{}' \mathcal{R}\ \mathbb{M}$ since $\mathbb{M} \xrightarrow{\tau} \mathbb{M}$).

When $\mathsf{N}_s \mathcal{R}\ \mathbb{M}$ for a partial satisfaction $\mathcal{R}$, we say N_s *satisfies* $\mathbb{M}$, writing $\models \mathsf{N}_s : \mathbb{M}$.

Note that, the transitions in the *partial network* are defined in Figure 5.4, which are invisible globally because a partial network does not include the global queue, which makes the transitions globally visible. Thus we use $\xrightarrow{\ell}$ for representing that there is a local process taking action ℓ in N_s instead of using $\xrightarrow{\ell}_{\mathrm{g}}$. Remember, the visible actions can be and can only be observed in the global transport.

Definition 7.4.26 (conformance to a configuration). Assume a network $\mathsf{N} \equiv \mathsf{N}_s \parallel H$ is given. Define N conforms to $\mathbb{M}; H$ when:

[1] The key ideas and concepts of satisfaction for the partial network come from Dr. Kohei Honda. The current version is the result after many discussions and revisions from our works. They also belong to the paper (20) which has been accepted and is going to be published.

1. H is empty, $\models \mathsf{N}_s : \mathbb{M}$ and $\mathbb{M}$ is consistent, or
2. H is not empty, and the following conditions hold
   (a) $\models \mathsf{N}_s : \mathbb{M}$,
   (b) all messages in H are receivable to N_s, and
   (c) as $\mathbb{M}; H \xrightarrow{\ell_1..\ell_n}_{\mathrm{g}} \mathbb{M}'; \emptyset$ so that $\mathsf{N}_s \parallel H \xrightarrow{\ell_1..\ell_n}_{\mathrm{g}} \mathsf{N}_s{}' \parallel \emptyset$ where each $\ell_i, i = \{1, ..., n\}$ is an input, $\mathbb{M}'$ is consistent.

Compared to the definition of global observable transition in Definition 7.2.13, which is for the *monitored* network, here define the global network transition for the *un-monitored* network.

Definition 7.4.27 (global network transition). Assume N is an un-monitored network and N conforms to $\mathbb{M}; H$. $\mathbb{M}; H \xrightarrow{\ell}_{\mathrm{g}} \mathbb{M}'; H'$ if and only if $\mathsf{N} \xrightarrow{\ell}_{\mathrm{g}} \mathsf{N}'$.

This definition corresponds to the illustration of global observability $\xrightarrow{\ell}_{\mathrm{g}}$ in Section 7.2.4.3. When N, although it is not monitored, conforms to the specifications specified in monitors, they have the same behaviour: When N takes actions, $\mathbb{M}; H$ takes actions, and vise versa.

Theorem 7.4.28 (session fidelity). Assume configuration $\mathbb{M}; H$ is configurationally consistent and $\mathsf{N} \equiv \mathsf{N}_s \parallel H$ conforms to configuration $\mathbb{M}; H$. We say N satisfies session fidelity, if for any ℓ, we have $\mathsf{N} \xrightarrow{\ell}_{\mathrm{g}} \mathsf{N}'$ such that $\mathbb{M}; H \xrightarrow{\ell}_{\mathrm{g}} \mathbb{M}'; H'$, it holds that $\mathbb{M}'; H'$ is configurationally consistent and that N' conforms to $\mathbb{M}'; H'$.

Proof. The proof and auxiliary theorems are in Appendix B.4.

7.5 Global Environment $\mathfrak{E}$

The introduction of *global observables* offers a formal framework to semantically link local behaviour of processes to their global behaviour and to global invariants. Since, in distributed processes, the sending events and the receiving events are decoupled, and because external monitors can only observe asynchronously exchanged messages, the semantic account of global invariants (hence correctness of runtime verification) should take into account temporary discrepancies of the global view. When a sender sends a message correctly, its local view is updated; however, as the receiver has not yet received the message, we cannot update neither the global view nor the receiver's local view. To

prove the correctness of distributed runtime verification, we here propose to take into account the time lag between the sending and receiving events. Asynchrony also poses a challenge in the treatment of out-of-order asynchronous message monitoring, which we capture through specification-level permutations of actions.

7.5.1 Motivating Example

As Figure 7.7 shows, from the perspective of the global network, all *global behaviours* of monitored processes become unobservable. To analyse and state properties of monitored networks, we propose to observe the global behaviours through linking global specifications to global interactions among monitored networks, neglecting local interactions inside endpoints. We propose a notion of global observables and define it through two labelled transitions systems: One is for the network and another is for environments (which represent the abstraction of global specifications and message flows).

As for the network, the notation of global observable transition $\xrightarrow{\ell}_{\mathrm{g}}$ is applied to $N \xrightarrow{\ell}_{\mathrm{g}} N'$ for N without hiding, if there exists a monitor in N having transition ℓ (by the LTS defined in Figure 7.4). We state the global observables, the actions which can be globally observed, is the aggregation of what all monitors observe and what one can observe from the global queue. As for environments, we formalise global observable environments, ranged over by $\mathfrak{E}, \mathfrak{E}', \ldots$, in order to witness the legality of all messages in transit. A global observable environment includes pending messages together with global specifications (for ease of defining its LTS, we also include local specifications). The use of pending messages is motivated as follows.

In the monitored network, we expect that all endpoint processes follow exactly what global specifications G define. This is the reason that we state that: For all sending and receiving actions, they are dual in the sending-side and receiving-side monitors when the collection of monitors (i.e. the collection of specifications) is consistent (Definition 7.4.12). However, while there are messages in transit, the consistency of monitors may fail: If a message has been sent from a local monitored process, there is no more monitor holds this sending action; but, since this message is in transit, the monitor at the receiver-side still holds the receiving action waiting for the message. Then the sender-side and the receiver-side monitors are not consistent. Here we propose that, the monitor at the receiver-side needs to be compensated not by other monitor specifications

but by the message in transit in the global queue. The following example illustrates the situation we consider.

Example 7.5.1. Assume a simple global protocol specifies that:

$$p_1 \to p_2 : (x : \mathsf{int})\langle\!\langle x > 5 \rangle\!\rangle.G_2$$

At the beginning, the specification of the sender-side monitor is

$$s[p_1]^\bullet : p_2!(x : \mathsf{int})\langle\!\langle x > 5 \rangle\!\rangle.G_2 \upharpoonright p_1$$

and the one of the receiver-side is

$$s[p_2]^\bullet : p_1?(x : \mathsf{int})\langle\!\langle x > 5 \rangle\!\rangle.G_2 \upharpoonright p_2$$

When the sender-side monitor permits an outgoing message $s\langle p_1, p_2, 10\rangle$, (i.e., a legal sending action), its specification immediately changes to $s[p_1]^\bullet : G_2 \upharpoonright p_1$ for the next action; however, the specification of receiver-side monitor may not change because it is waiting for the message travelling in the global queue $H \cdot s\langle p_1, p_2, 10\rangle$; as long as the message does not arrive, the receiver-side monitor cannot change its specification. In this case, the receiver-side monitor knows the messages will certainly come by looking at the global queue.

7.5.2 The Syntax of $\mathfrak{E}$

In this section, we formalise the notion of *the global observable* which can witness the legality of all messages in transit. We call $\mathfrak{E}$ a *global environment*, its syntax is defined as follows:

$$\begin{array}{rcl} \mathfrak{E} & ::= & \Gamma, \Delta, \Theta \\ \Gamma & ::= & \emptyset \mid \Gamma, a : \mathtt{mode}(G[p]) \\ \Delta & ::= & \emptyset \mid \Delta, s[p] : T \mid \Delta, s[p]^\bullet : T \mid \Delta, s : \widetilde{mv} \\ \Theta & ::= & \emptyset \mid \Theta, s : G \\ mv & ::= & s\langle p, q, l\langle v\rangle\rangle \end{array}$$

where $\mathtt{mode} = \{\mathtt{I}, \mathtt{O}, \mathtt{IO}\}$, Γ is for shared environment, Δ is for session environment as the one in Figure 7.3, to which we add messages belonging to session s as $s : \widetilde{mv}$, to specify runtime pending messages carrying the information, and Θ is a global environment associating sessions to global specifications. All messages of different sessions model a global queue environment where each assignment is of the form $mv_1 \dots mv_n$, $n \geq 0$.

mv_i is a message of shape $s\langle p, p', l\langle v\rangle\rangle$. $\widetilde{mv}$ is a sequence of pending messages, such as $s_i\langle p_1, p_2, l_1\langle 10\rangle\rangle \cdot s_j\langle p_3, p_5, l_2\langle \text{“}hello''\rangle\rangle \cdot s_k\langle p_1, p_3, l_3\langle \mathsf{true}\rangle\rangle$. Thus the global observable $\mathfrak{E}$ is composed by the standard shared and session environments, global specifications, and global messages in the global queue.

7.5.3 The Semantics of $\mathfrak{E}$

Before introducing the LTS of $\mathfrak{E}$, define the projection of a sequence of messages to a particular role. The projection results in a sequence of projected messages sent from the role and indicates their destinations. The projected message is in the form $p!l(v)$, denoted by pm.

Definition 7.5.2 (projection of messages). The *projection of* $\widetilde{mv}$ *onto* p, written $\widetilde{mv} \upharpoonright p$, is defined as:

$$\widetilde{mv} \upharpoonright p = \begin{cases} p_2!l(v) \cdot \widetilde{mv}' \upharpoonright p & \text{if } \widetilde{mv} = s\langle p_1, p_2, l\langle v\rangle\rangle \cdot \widetilde{mv}' \text{ and } p = p_1 \\ \widetilde{mv}' \upharpoonright p & \text{if } \widetilde{mv} = s\langle p_1, p_2, l\langle v\rangle\rangle \cdot \widetilde{mv}' \text{ and } p \neq p_1 \\ \varepsilon & \text{if } \widetilde{mv} = \varepsilon \end{cases}$$

$\widetilde{mv} \upharpoonright p$ is defined as above because the messages in transit are the source to indicate (so that the receiver-side monitor can refer to) that the corresponding sending actions have taken place.

Definition 7.5.3 (retrieving a sub-T). Let $\mathsf{pm}, \mathsf{pm}', ..$ range over projected pending messages. Assume T and pm are given. We set:

$$T - \mathsf{pm} = \begin{cases} T_j\{v/x_j\} - \mathsf{pm}' & \text{if } T \curvearrowright p!\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.T_i\}_{i\in I} \\ & \text{and } \mathsf{pm} = p!l_j(v) \cdot \mathsf{pm}', j \in I \\ T & \text{if } \mathsf{pm} = \varepsilon \\ \text{undefined} & \text{otherwise} \end{cases}$$

where I is the set of indexes.

In Definition 7.5.3, pm indicates the floating messages which are sent from the endpoint whose monitor has specification T. In other words, pm and T are specified at the same endpoint.

Define $\mathfrak{E} \xrightarrow{\ell}_{\mathrm{g}} \mathfrak{E}'$ when the session environment in $\mathfrak{E}$ allows ℓ and, after the configuration of some specification or the global queue in $\mathfrak{E}$ changes, results in $\mathfrak{E}'$. Figure 7.11 defines the labelled transition system for $\mathfrak{E}$.

In [$\mathfrak{E}$-BCH], $s : G$ means that session s obeying global specification G. Note that, T is not simply the projection from G (i.e. T may not be $G \upharpoonright p_2$) but

$$\begin{array}{rcl} T & = & G \upharpoonright p_2 - (s\langle p_1, p_2, l_j\langle v\rangle\rangle \cdot \widetilde{mv}) \upharpoonright p_2 \\ & = & G \upharpoonright p_2 - \widetilde{mv} \upharpoonright p_2, \end{array}$$

which is obtained by removing all *outputted actions* produced by p_2 (e.g., $s\langle p_2, p, l\langle v\rangle\rangle$ for some p) from all actions of p_2 (i.e., $G \upharpoonright p_2$). T_j is similarly represented as:

$$T_j \quad = \quad (G_j \upharpoonright p_2)\{v/x_j\} - \widetilde{mv} \upharpoonright p_2$$

obtained by removing outputted actions of p_2 from all actions in G_j of p_2 and replacing x_j by v in $G_j \upharpoonright p_2$. [$\mathfrak{E}$-BCHN] is similar to the rule [BRAN] defined in Figure 7.4, when a name a is received by $s[p_1, p_2]?l(a)$, $a : \mathtt{0}(G'[p])$ is added into the environment.

The reason of adopting a more complex method instead of simply applying $G \upharpoonright p_2$, is to obtain the appropriate endpoint specification considering the asynchrony nature of interactions while messages may not be received by recipients immediately due to routing delays. We use the following example to show this situation.

Example 7.5.4. Assume a global specification

$$\begin{array}{l} p_1 \to q_1 : (x_1 : \mathsf{int})\langle\!\langle x_1 > 0\rangle\!\rangle.p_1 \to q_2 : (x_2 : \mathsf{int})\langle\!\langle x_2 > 1\rangle\!\rangle. \\ q_3 \to p_1 : (x_3 : \mathsf{int})\langle\!\langle x_3 > 2\rangle\!\rangle.\mathsf{end} \end{array} \tag{7.1}$$

Obviously, as p_1 is active, the local specification of monitor at p_1 is

$$s[p_1]^\bullet : q_1!(x_1 : \mathsf{int})\langle\!\langle x_1 > 0\rangle\!\rangle.q_2!(x_2 : \mathsf{int})\langle\!\langle x_2 > 1\rangle\!\rangle.q_3?(x_3 : \mathsf{int})\langle\!\langle x_3 > 2\rangle\!\rangle.\mathsf{end} \tag{7.2}$$

It is possible that participants interact in the following order: (I) q_3 firstly sends message $s\langle q_3, p_1, \langle 3\rangle\rangle$ to p_1, (II) then p_1 sends messages to q_1 and q_2 with $s\langle p_1, q_1, \langle 1\rangle\rangle$ and $s\langle p_1, q_2, \langle 2\rangle\rangle$ respectively. Note that, as the first interaction happens, p_1 may not receive

$$\mathfrak{E} \xrightarrow{\tau}_{\mathrm{g}} \mathfrak{E} \qquad [\mathfrak{E}\text{-TAU}]$$

$$\frac{\mathfrak{E} \vdash v : S_j \quad A_j\{v/x_j\} \downarrow \mathsf{true} \quad G \curvearrowright p_1 \to p_2 : \{l_i(x_i{:}S_i)\langle\!\langle A_i \rangle\!\rangle.G_i\}_{i\in I}}{\mathfrak{E}, s : G, s[p_2]^\bullet : T, s : s\langle p_1, p_2, l_j\langle v\rangle\rangle \cdot \widetilde{mv} \xrightarrow{s[p_1,p_2]?l_j(v)}_{\mathrm{g}} \mathfrak{E}, s : G_j\{v/x_j\}, s[p_2]^\bullet : T_j, s : \widetilde{mv}} \qquad [\mathfrak{E}\text{-BCH}]$$

$$\frac{\begin{array}{c} S_j = \mathtt{mode}(G'[p]),\ a \notin A_j, \\ A_j \downarrow \mathsf{true} \quad G \curvearrowright p_1 \to p_2 : \{l_i(x_i{:}S_i)\langle\!\langle A_i \rangle\!\rangle.G_i\}_{i\in I} \end{array}}{\mathfrak{E}, s : G, s[p_2]^\bullet : T, s : s\langle p_1, p_2, l_j\langle a\rangle\rangle \cdot \widetilde{mv} \xrightarrow{s[p_1,p_2]?l_j(a)}_{\mathrm{g}} \mathfrak{E}, s : G_j\{a/x_j\}, a : \mathtt{0}(G'[p]), s[p_2]^\bullet : T_j, s : \widetilde{mv}} \qquad [\mathfrak{E}\text{-BCHN}]$$

$$\frac{\mathfrak{E} \vdash v{:}S_j \quad A_j\{v/x_j\} \downarrow \mathsf{true} \quad T \curvearrowright p_2!\{l_i(x_i : S_j)\langle\!\langle A_i \rangle\!\rangle.T_i\}_{i\in I}}{\mathfrak{E}, s[p_1]^\bullet{:}T, s{:}\widetilde{mv} \xrightarrow{s[p_1,p_2]!l_j\langle v\rangle}_{\mathrm{g}} \mathfrak{E}, s[p_1]^\bullet{:}T_j\{v/x_j\}, s{:}\widetilde{mv}\cdot s\langle p_1, p_2, l_j\langle v\rangle\rangle} \qquad [\mathfrak{E}\text{-SEL}]$$

$$\frac{\mathfrak{E} \vdash a{:}S_j, a \notin A_j,\ A_j \downarrow \mathsf{true} \quad T \curvearrowright p_2!\{l_i(x_i : S_j)\langle\!\langle A_i \rangle\!\rangle.T_i\}_{i\in I}}{\mathfrak{E}, s[p_1]^\bullet{:}T, s{:}\widetilde{mv} \xrightarrow{s[p_1,p_2]!l_j\langle a\rangle}_{\mathrm{g}} \mathfrak{E}, s[p_1]^\bullet{:}T_j\{a/x_j\}, s{:}\widetilde{mv}\cdot s\langle p_1, p_2, l_j\langle a\rangle\rangle} \qquad [\mathfrak{E}\text{-SELN}]$$

$$\frac{\mathtt{im} \in \{\mathtt{I}, \mathtt{IO}\},\ a \notin \mathsf{dom}(\mathfrak{E})}{\mathfrak{E} \xrightarrow{\mathsf{new}\ a:\mathtt{im}(G[p])}_{\mathrm{g}} \mathfrak{E}, a{:}\mathtt{im}(G[p])} \qquad [\mathfrak{E}\text{-NEW A}]$$

$$\mathfrak{E}, s[p]{:}T \xrightarrow{\mathsf{join}(s[p])}_{\mathrm{g}} \mathfrak{E}, s[p]^\bullet{:}T \qquad [\mathfrak{E}\text{-JOIN}]$$

$$\frac{\begin{array}{c} s \notin \mathsf{dom}(\mathfrak{E}),\ \mathtt{om} \in \{\mathtt{O}, \mathtt{IO}\}, \\ p_i \in \mathit{role}(G),\ G \text{ is well-formed},\ \forall k, j \in I, p_k \neq p_j\ (k \neq j) \end{array}}{\mathfrak{E}, a{:}\mathtt{om}(G[p]) \xrightarrow{\overline{a}(s[p]:G)}_{\mathrm{g}} \mathfrak{E}, a{:}\mathtt{om}(G[p]), s{:}G, \{s[p_i] : (G \upharpoonright p_i)\}_{p_i\in \mathit{role}(G)}, s : \emptyset} \qquad [\mathfrak{E}\text{-REQ-B}]$$

$$\frac{s \in \mathfrak{E},\ G \upharpoonright p = T,\ \mathtt{om} \in \{\mathtt{O}, \mathtt{IO}\},\ \mathtt{im} \in \{\mathtt{I}, \mathtt{IO}\}}{\mathfrak{E}, a{:}\mathtt{om}(G[p]), a{:}\mathtt{im}(G[p]), s[p]{:}T \xrightarrow{\overline{a}\langle s[p]:G\rangle}_{\mathrm{g}} \mathfrak{E}, a{:}\mathtt{om}(G[p]), a{:}\mathtt{im}(G[p]), s[p]{:}T} \qquad [\mathfrak{E}\text{-REQ-F}]$$

$$\frac{s \in \mathfrak{E},\ G \upharpoonright p = T,\ \mathtt{im} \in \{\mathtt{I}, \mathtt{IO}\}}{\mathfrak{E}, a{:}\mathtt{im}(G[p]), s[p]{:}T \xrightarrow{a(s[p]:G)}_{\mathrm{g}} \mathfrak{E}, a{:}\mathtt{im}(G[p]), s[p]{:}T} \qquad [\mathfrak{E}\text{-ACC-B}]$$

$$\frac{s \in \mathfrak{E},\ G \upharpoonright p = T,\ \mathtt{im} \in \{\mathtt{I}, \mathtt{IO}\}}{\mathfrak{E}, a{:}\mathtt{im}(G[p]), s[p]{:}T \xrightarrow{a\langle s[p]:G\rangle}_{\mathrm{g}} \mathfrak{E}, a{:}\mathtt{im}(G[p]), s[p]{:}T} \qquad [\mathfrak{E}\text{-ACC-F}]$$

Figure 7.11: The Labelled Transition System of $\mathfrak{E}$

this message immediately. Assume before this message arrives, p_1 has sent out messages $s\langle p_1, q_1, \langle 1\rangle\rangle$ and $s\langle p_1, q_2, \langle 2\rangle\rangle$. Therefore, the global queue has

$$s\langle q_3, p_1, \langle 3\rangle\rangle \cdot s\langle p_1, q_1, \langle 1\rangle\rangle \cdot s\langle p_1, q_2, \langle 2\rangle\rangle$$

and the global specification *maintains* as Equation (7.1) because no message has been received yet. Note that, the current specification of monitor at p_1 is $s[p_1]^\bullet : q_3?(x_3 : \mathsf{int})\langle\!\langle x_3 > 2\rangle\!\rangle.\mathsf{end}$ since it has done two output actions. In such case, $G \upharpoonright p_1$, which is still in Equation (7.2) (because G does not change), cannot reflect the reality of p_1 due to the asynchrony nature.

Continue with rule [$\mathfrak{E}$-BCH]. It says that, if a value is typed with S_j and satisfies A_j, and G has a corresponding interaction up to permutations, it allows $s\langle p_1, p_2, l_j\langle v\rangle\rangle$ to be received, resulting in new local/global specifications.

$G \curvearrowright G'$ and $T \curvearrowright T'$ are defined in Definitions 6.6.5 and 6.6.8. The relation $G \curvearrowright G'$ says specification G can be permuted to G', so that what is not *apparently* an active action in G becomes active in G' modulo permutation. For example, if both $Buyer$ and $Broker$ are sending a message to $Seller$:

$$G = Buyer \to Seller : (x : \mathsf{int}).Broker \to Seller :: (x : \mathsf{string}).\mathsf{end}$$

then $Broker$'s message may as well arrive at $Seller$ first, hence we permute this to

$$G' = Broker \to Seller : (x : \mathsf{string}).Buyer \to Seller : (x : \mathsf{int}).\mathsf{end}$$

where $G \curvearrowright G'$ and G' is ready to check an appropriate message from $Broker$ to $Seller$.

Rule [$\mathfrak{E}$-SEL] states that when $\mathfrak{E}$ approves an output, its global specification is unchanged but put a message to the global queue, indicating that an interaction has partially happened by an output, but the whole interaction is not completed. And, since $T \curvearrowright p_2!\{l_i(x_i : S_j)\langle\!\langle A_i\rangle\!\rangle.T_i\}_{i\in I}$, after output action $s[p_1, p_2]!l_j\langle v\rangle$ takes place, the chosen label l_j leads configuration of local specification to be $T_j\{v/x_j\}$ as v replaces x_j for the next interaction. Rule [$\mathfrak{E}$-SELN] is defined similarly to the rule [SELN] defined in Figure 7.4. [$\mathfrak{E}$-NEW A] says if a shared name a is new to $\mathfrak{E}$, then $\mathfrak{E}$ adds this new shared channel. [$\mathfrak{E}$-JOIN] makes a specified session-role $s[p]$ become active. [$\mathfrak{E}$-REQ-B] says if a session s is new to $\mathfrak{E}$, $\mathfrak{E}$ adds the global queue belonging to session s as $s : \emptyset$, the global specification of s as $s : G$, and the session environment which is generated by

the projections of G to all roles in G, write $\{s[p_i]:(G \upharpoonright p_i)\}_{p_i \in role(G)}$, every p_i of which involves in s. Since $\mathfrak{E}$ watches the global environment containing the information of all local processes, as free request or bound/free accept happens at an endpoint, it does not affect the global environment. [$\mathfrak{E}$-REQ-F] and [$\mathfrak{E}$-ACC-B/F] state this fact. Note that, for [$\mathfrak{E}$-REQ-F] (or [$\mathfrak{E}$-ACC-B/F]), if the left-hand side environment violates the rule, the rule [$\mathfrak{E}$-TAU] is applied.

7.5.4 The Typing Rules of $\mathfrak{E}$

In the following rules, we define $\mathfrak{E} \vdash_{\mathrm{g}} N$ as a network N is proved under the global observable system, i.e. $\vdash_{\mathrm{g}}$, by environment $\mathfrak{E}$.

$$\frac{\mathcal{M} = \Gamma', \Delta \quad \Gamma' \subset \Gamma}{\Gamma, \Delta, \mathfrak{E} \vdash_{\mathrm{g}} \mathcal{M}[P]}[\mathfrak{E}\text{-M}] \qquad \frac{\Gamma \vdash_{\mathrm{g}} a{:}\mathtt{im}(G[p]), a{:}\mathtt{O}(G[p])}{\Gamma, \mathfrak{E} \vdash_{\mathrm{g}} H \cdot \overline{a}\langle s[p] : G\rangle}[\mathfrak{E}\text{-PM1}]$$

$$\frac{H = \emptyset}{\Gamma, s : H \vdash_{\mathrm{g}} H}[\mathfrak{E}\text{-PM2}] \qquad \frac{\Gamma, s : \widetilde{m v} \vdash_{\mathrm{g}} H}{\Gamma, s : \widetilde{m v} \cdot s\langle p, p', l\langle v\rangle\rangle \vdash_{\mathrm{g}} H \cdot s\langle p, p', l\langle v\rangle\rangle}[\mathfrak{E}\text{-PM3}]$$

$$\frac{\Gamma, \Delta_i, \Theta \vdash_{\mathrm{g}} N_i \quad (i = 1, 2) \quad \mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Delta_2) = \emptyset}{\Gamma, \Delta_1, \Delta_2, \Theta \vdash_{\mathrm{g}} N_1 \parallel N_2}[\mathfrak{E}\text{-PAR}]$$

$$\frac{\mathfrak{E} \vdash_{\mathrm{g}} N \qquad \mathfrak{E} \text{ is coherent}}{\mathfrak{E} \vdash N}[\mathfrak{E}\text{-CO}]$$

Every rule is explained as follows:

[$\mathfrak{E}$-M] As the information about monitor $\mathcal{M}$ is known by the global environment, the global environment approves the monitored process $\mathcal{M}[P]$.

[$\mathfrak{E}$-PM1] As the shared environment has $a : \mathtt{im}(G[p])$, which indicates that a is capable of playing role p under the specification G, the invitation message $\overline{a}\langle s[p] : G\rangle$, produced by either bound request $\overline{a}(s[p] : G)$ or free request $\overline{a}\langle s[p] : G\rangle$, is proved by the global environment $\Gamma, \mathfrak{E}$.

[$\mathfrak{E}$-PM2] As the global queue is empty, it is always valid under $\Gamma, s : H$, for any Γ.

[$\mathfrak{E}$-PM3] As the global environment $\Gamma, s : \widetilde{m v}$ proves the pending messages H, the additional message $s\langle p_1, p_2, l\langle v\rangle\rangle$ can be proved by $\Gamma, s : \widetilde{m v} \cdot s\langle p_1, p_2, l\langle v\rangle\rangle$ where the message environment contains the additional message.

[$\mathfrak{E}$-PAR] As, for $i = 1, 2$, the global environment Γ, Δ_i, Θ proves the network N_i and Δ_1 and Δ_2 has no common session roles or common message environment, then the global environment $\Gamma, \Delta_1, \Delta_2, \Theta$ proves the composed network $N_1 \parallel N_2$.

[$\mathfrak{E}$-CO] As the network N can be ensured (i.e. proved) under the system of the global observable environment, we say the network *conforms* to the global observable $\mathfrak{E}$. The coherence of a network to the global observable $\mathfrak{E}$ is introduced in the next section.

7.5.5 The Coherence of $\mathfrak{E}$

Definition 7.5.5. As long as s : end is reached, session s is finished and everything related to s is deleted immediately from $\mathfrak{E}$. Whenever $s : G$ where $G = \mathsf{end}$, $s : G \notin \mathfrak{E}$.

Coherence of $\mathfrak{E}$ is defined similarly to the one of the network, taking care of the pending messages in the global queue.

Definition 7.5.6 ($\mathfrak{E}$ coherence). Let $\mathfrak{E} = \Gamma, \Delta, \Theta$. Then we say $\mathfrak{E}$ is *coherent* when all of the following conditions hold:

1. $\forall a$, $a : \mathtt{om}(G[p]) \in \Gamma$ implies that $\exists \Gamma', a : \mathtt{im}(G[p]) \in \Gamma'$.
2. If $s[p] \in \Delta$, then $s{:}G \in \Theta$ and $p \in \mathit{role}(G)$.
3. $\forall\, p \in \mathit{role}(G)$, if $s{:}G \in \Theta$ and $s : \widetilde{m}v \in \Delta$, then $\exists\, T$ such that $G \upharpoonright p - \widetilde{m}v \upharpoonright p = T$ and $\Delta(s[p]^\circ) \curvearrowright T$.
4. If $s : G \in \Theta$ and $s : \widetilde{m}v_1 \cdot s\langle p, q, l_j\langle v\rangle\rangle \cdot \widetilde{m}v_2 \in \Delta$ and $\widetilde{m}v_1$ does not suppress $s\langle p, q, l_j(v)\rangle$, then
$$G \curvearrowright p \to q : \{l_i(x_i : S_i)\langle\!\langle A_i\rangle\!\rangle.G'_i\}_{i\in I}$$
where $A_j\{v/x_j\} \downarrow \mathsf{true}$.

The first rule is standard. It says whenever there is an output mode for capability of playing role p specified under G, there exists its corresponding input mode in some shared environment Γ'.

The second rule of $\mathfrak{E}$-coherence says: Every session-role $s[p]$ comes from a global specification, which defines this session s and all roles, i.e. p, in this session.

In the third rule, $G \upharpoonright p$ represents all interactions defined in G at p and the order of actions that p should fire; similarly, $\widetilde{m}v \upharpoonright p$ indicates the targets (i.e. destinations) of

pending messages sent from p, and $G \upharpoonright p - \widetilde{mv} \upharpoonright p$ together is the left specification after p outputs to targets in $\widetilde{mv} \upharpoonright p$. This rule can be read as: For every role in a session, its fully defined actions are exactly the summation of the "happened" actions, which are currently recorded in the global queue, and the "going-to-happen" actions.

The forth rule states the relation between pending messages and its global specification. The existence of a pending message means that the corresponding output has happened, but input has not yet taken place. This rule says: The existence of a pending message implies (I) its corresponding interaction should be defined in global specification, and (II) this interaction can be permuted to the *top* of its global specification with minimum number of unit permutations (i.e. it is not suppressed by any interaction sequenced ahead of it).

Proposition 7.5.7 (conformance w.r.t. the global observables). If $\mathfrak{E}$ is coherent and $\mathfrak{E} \xrightarrow{\ell}_{\mathrm{g}} \mathfrak{E}'$, then $\mathfrak{E}'$ is coherent.

Proof. The proof is in Appendix B.5.

Definition 7.5.8 ($\mathfrak{E} \vdash N$). We write $\mathfrak{E} \vdash N$ to state that network N conforms to $\mathfrak{E}$ as long as

1. $\mathfrak{E}$ is coherent, and
2. the shared and session environments in $\mathfrak{E}$ come from the *monitors* in N, and
3. the message environment in $\mathfrak{E}$ comes from the pending messages in the global queue of N.

If $\mathfrak{E}$ is coherent and $\mathfrak{E} \vdash N$, then N is coherent. Therefore, with the formalism of global observer, another viewpoint of the property of session fidelity is:

Theorem 7.5.9 (session fidelity w.r.t. $\mathfrak{E}$). If $\mathfrak{E} \vdash N$ and $N \xrightarrow{\ell}_{\mathrm{g}} N'$ then $\mathfrak{E} \xrightarrow{\ell}_{\mathrm{g}} \mathfrak{E}'$ such that $\mathfrak{E}' \vdash N'$.

Proof. It is proved in Appendix B.5.

The global observable $\mathfrak{E}$ maintains all specifications, including the global and the local ones, and together with pending messages. Whenever N is coherent, we can construct a (coherent) $\mathfrak{E}$ such that $\mathfrak{E} \vdash N$. The version of session fidelity w.r.t $\mathfrak{E}$ states that, global interactions in a coherent network never violate the expected network-wise global specifications: The former follows the latter step by step.

8

Specifying Stateful Asynchronous Properties for Distributed Programs

Overview Chapter 8 is a full version of the published paper "Specifying Stateful Asynchronous Properties for Distributed Programs" (40). In this chapter, the theorems and properties of *stateful* specifications for *dynamic observations* among communicating processes are introduced. They are based on the condition that observations are done asynchronously, where a semantic problem in specifications for distributed systems arises. Consider a general situation described as follows: When an observer (e.g. a trusted monitor) is located at an observee, the order of the observee's actions the observer sees is exactly the same as what happens at the observee. However, when an observer sits outside the observee, e.g. remotely from the observee, the order of actions that she observes may not be as same as what happens at the observee. The former kind of observation is *synchronous*, and the latter kind is *asynchronous*. Although the synchronous observation precisely captures the behaviour of the observee, in real-life distributed systems, asynchronous observation is the norm and often a necessity. To make a remote observer be able to properly verify behaviours of processes against a non-trivial stateful specification, this chapter characterises and validates *asynchronously verifiable specifications*, or simply called *asynchronous specifications*. We also propose and prove the properties for them.

8.1 Motivating Example

Before formally introducing the syntax and semantics of specifications, this section motivates key ideas through simple examples. Our specification language is based on multi-party session types (18, 86) with annotations by logical formulae, extending (22) with endpoint (local) states. Our focus is on visible asynchronous interactions rather than internal actions. We treat three examples. The first motivates the use of state in protocol specifications through a typical business scenario. The second introduces the semantic issue in specifications in the presence of asynchrony through a simplest possible example. The third example illustrates the use of sets in specifications for asynchrony, linking the specification to the corresponding traces, adumbrating our semantic analysis later.

8.1.1 Using State in Specifications

The first example motivates the use of endpoint state in a stateful protocol, which is a part of a specification. Consider the following purchase-invoice scenario:

(step 1) *Buyer* sends a *product name* (denoted by variable *PName*) to *Seller*, then *Seller* replies with its *price* (denoted by variable *Price*), and *Buyer* decides whether to purchase (then go to step 2) or not (then terminate). We assume shipping is done independently.

(step 2) *Seller* sends *Buyer* an *invoice* for the purchased product.

This scenario can be realised as a single protocol (22, 39, 86) between *Buyer* and *Seller*, consisting of a series of a few message exchanges. The scenario can also be realised using *two* protocols, one for each step. This form has a merit in flexibility: For example, when *Buyer* and *Seller* finish step 1, both can terminate that transaction. Then an invoice can be issued any time later. Example 8.1.1 presents protocols with logical annotations following the framework of representing one scenario with two separated protocols.

Example 8.1.1 (SP for a cross-session Purchase-and-Invoice scenario).

$$
\begin{array}{rcl}
G_{\mathsf{pcs}} & = & B \to S : \mathit{Request}(\mathit{PName} : \mathsf{string}). \\
& & S \to B : \mathit{Confirm}(\mathit{PNameConf} : \mathsf{string}, \mathit{Price} : \mathsf{int}) \\
& & \qquad \langle\!\langle \mathit{PNameConf} = \mathit{PName} \ \cap \ \mathit{Price} \geq 0; \varepsilon \rangle\!\rangle \\
& & \qquad \langle\!\langle \mathsf{true}; \varepsilon \rangle\!\rangle. \\
& & B \to S : \{ \mathit{OK}(\mathit{UserID} : \mathsf{int}) \langle\!\langle \mathit{UserID} \neq 0; \varepsilon \rangle\!\rangle \langle\!\langle \mathsf{true}; \varepsilon \rangle\!\rangle. \\
& & \qquad S \to B : (\mathit{PNo} : \mathsf{int}) \\
& & \qquad\quad \langle\!\langle \mathit{PNo} \notin \mathsf{dom}(\mathbf{PLog}); \\
& & \qquad\qquad \mathbf{PLog} := \mathbf{PLog} \cup \{ \mathit{PNo} \mapsto (\mathit{UserID}, \mathit{PName}, \mathit{Price}) \} \rangle\!\rangle \\
& & \qquad\quad \langle\!\langle \mathsf{true}; \varepsilon \rangle\!\rangle. \\
& & \qquad \mathsf{end} \\
& & \quad \mathit{KO}().\mathsf{end} \} \\
G_{\mathsf{ivc}} & = & S \to B : (\mathit{PNo} : \mathsf{string}, \mathit{Invoice} : \mathsf{int}) \\
& & \qquad \langle\!\langle \mathit{PNo} \in \mathsf{dom}(\mathbf{PLog}) \ \cap \mathit{Invoice} = \mathbf{PLog}(\mathit{PNo}) \rangle\!\rangle \langle\!\langle \mathsf{true}; \varepsilon \rangle\!\rangle.\mathsf{end}
\end{array}
$$

Above G_{pcs} and G_{ivc} denote the global *stateful protocols*, or SPs from now on for short, corresponding to Steps 1 and 2. Each specifies the behaviour which the participants, S (denoting seller) and B (denoting buyer), should realise at each session. $\langle\!\langle ...; ... \rangle\!\rangle \langle\!\langle ...; ... \rangle\!\rangle$ are the obligations for sender (the former) and receiver (the latter), respectively. Note that $\langle\!\langle \mathsf{true}; \varepsilon \rangle\!\rangle$ means no obligation and no state change. We will formally introduce their syntax in Section 8.3. In this example, the state of S, represented by the field **PLog** (the Purchase Log, which we consider to be a key-value store, mapping distinct keys to values), links the two protocols. Both protocols can be read intuitively. First, in G_{pcs},

1. B firstly sends a request (*Request* is an operator name), with the message value *PName* of type string, which is a product name.

2. S confirms by sending the same product name and its price, where the latter should be a non-negative integer as annotated.

3. If B says *OK* by sending its identity, then (in practice, after authenticating the identity) S sends back a *fresh* purchase number *PNo* which should be fresh, i.e. it should not be in the domain of **PLog**. As a result, this new key and the corresponding information is added to **PLog**. On the other hand, if B says *KO* (not *OK*), then the conversation terminates.

As may be seen from this example, our protocols use a local state to record an abstraction of preceding interactions across sessions, and to constrain future behaviours using this record. Its ultimate aim is to specify visible behaviours of a process: Thus the stipulated state does not (have to) come from an actual state of a process, i.e. we may call it a "ghost state" following JML (1).

8.1.2 Synchrony v.s. Asynchrony for Stateful Specifications

The next example illustrates the central topic of this chapter, asynchrony in specifications, showing how a specification can be "too synchronous" for asynchronous observations. We focus on a part of the previous example, and, as well use stateful protocols, which compose specifications, to indicate the asynchrony issue. The purchase number allocator S will, upon a request from a buyer B at each session, issue a booking number incrementing the previously issued one: S issues e.g. 1, 2, 3, ... in a sequence of sessions. Example 8.1.2 (a) shows a protocol G_{sync} for such interactions, specifying a simple behaviour that the participants, S and B, should realise at each session. $\mathbf{c}$ is a local state of S, denoting the next booking number.

Example 8.1.2 (Stateful Protocol of purchase number allocator: synchronous v.s. asynchronous).

(a) synchronous spec

$$
\begin{array}{lcl}
G_{\mathsf{sync}} & = & B \to S : \mathrm{req}(\varepsilon). \\
 & & S \to B : \mathrm{ans}(x : \mathsf{int}) \\
 & & \quad \langle\!\langle x = \mathbf{c};\ \mathbf{c} := \mathbf{c} + 1 \rangle\!\rangle \langle\!\langle \mathsf{true}; \varepsilon \rangle\!\rangle. \\
 & & \mathsf{end}
\end{array}
$$

(b) asynchronous spec

$$
\begin{array}{lcl}
G_{\mathsf{async}} & = & B \to S : \mathrm{req}(\varepsilon). \\
 & & S \to B : \mathrm{ans}(x : \mathsf{int}) \\
 & & \quad \langle\!\langle x \notin \mathbf{c};\ \mathbf{c} := \mathbf{c} \cup \{x\} \rangle\!\rangle \langle\!\langle \mathsf{true}; \varepsilon \rangle\!\rangle. \\
 & & \mathsf{end}
\end{array}
$$

In the first line of G_{sync}, B (for buyer) requests S (for seller) a purchase number by sending $\mathrm{req}(\varepsilon)$, where ε means there is no message value in this request. In the second line, an integer x is sent from S to B, for which $\langle\!\langle x = \mathbf{c}; \mathbf{c} := \mathbf{c} + 1 \rangle\!\rangle$ specifies the *obligation* for S, while no obligation i.e. $\langle\!\langle \mathsf{true}; \varepsilon \rangle\!\rangle$ for B. The first part "$x = \mathbf{c}$" says that x should be equal to $\mathbf{c}$. The second part "$\mathbf{c} := \mathbf{c} + 1$" says that, after sending, S will increase $\mathbf{c}$ by 1, which constrains further behaviours of S in later sessions.

G_{sync} is an example of a SP which makes sense synchronously but *not* asynchronously. It seems an intuitively sensible stateful protocol: However, if a remote observer is far away, even if S *actually* sends the series of booking numbers $1, 2, 3, 4, \ldots$ in this order, they may arrive at the observer as e.g. $2, 4, 1, 3, \ldots$, under the assumption that the order

of messages belonging to *distinct* sessions may not be preserved, which is a practical assumption. In particular, note this remote observer will consider S as being *ill-behaved with respect to* G_{sync}: The correctness for S (which is synchronous) and the correctness for its observer (which is asynchronous) are not congruent.

As a remedy, we present G_{async} in Example 8.1.2 (b), which is intended for asynchronous observation. We use the *set* of booking numbers: $\mathbf{c}$, whose type is a set of integers, corresponds to **PLog** in Example 8.1.1. The new stateful protocol just says, in brief, that "S always sends a fresh number". If the behaviour of S satisfies this condition at S, then even though messages from S may arrive out-of-order, the remote observer can verify that they are correct w.r.t. G_{async}, so that the actions of S and their asynchronous observation by a remote observer coincide. We shall later verify this statement formally.

8.1.3 Capturing Causality Using Sets in Stateful Specifications

While G_{async} gives a reasonable stateful protocol, it is not the strongest possible one if our target is the server who issues booking numbers incrementally based on the previous number. For example, if the same buyer sequentially repeats a series of request-reply sessions, that buyer (and an observer sitting in-between) will observe $1, 2, 3, 4$, but this point is not captured by G_{async}.

Example 8.1.3 (a refinement of G_{async}).

$$\begin{array}{rcl} G_{\mathsf{assign}} & = & B \to S : \mathrm{req}(\varepsilon)\langle\!\langle \mathsf{true}; \varepsilon \rangle\!\rangle \langle\!\langle \mathsf{true}; \mathbf{t} := \mathbf{t}+1, \mathbf{c} := \mathbf{c} \cup \{\mathbf{t}\} \rangle\!\rangle. \\ & & S \to B : \mathrm{ans}(x : \mathsf{int}) \langle\!\langle x \in \mathbf{c};\ \mathbf{c} := \mathbf{c} \setminus \{x\} \rangle\!\rangle \langle\!\langle \mathsf{true}; \varepsilon \rangle\!\rangle. \\ & & \mathsf{end} \end{array}$$

G_{assign} in Example 8.1.3 is a refinement of G_{async} in Example8.1.2 so that, while still being suitable for asynchronous observations, can capture a stronger causal constraint. It uses two states: $\mathbf{t}$, a counter, and $\mathbf{c}$, a collection of valid numbers to be issued. $\mathbf{t}$ and $\mathbf{c}$ are incremented when receiving a request, while the sent value is taken off from $\mathbf{c}$.

Assume the server issues the booking numbers starting from 1. The intuition behind the construction is as follows:

1. If S receives n requests, as a whole, the numbers which can be issued are among $\{1, 2, .., n\}$.

Traces of permitted actions with the corresponding state change.

cases	1st	2nd	3rd	4th
(I) actions	$s_1[B,S]?\text{req}(\varepsilon)$	$s_2[B,S]?\text{req}(\varepsilon)$	$s_1[S,B]!\text{ans}\langle 1\rangle$	$s_2[S,B]!\text{ans}\langle 2\rangle$
	$s_2[B,S]?\text{req}(\varepsilon)$	$s_1[B,S]?\text{req}(\varepsilon)$	$s_1[S,B]!\text{ans}\langle 1\rangle$	$s_2[S,B]!\text{ans}\langle 2\rangle$
	$s_1[B,S]?\text{req}(\varepsilon)$	$s_2[B,S]?\text{req}(\varepsilon)$	$s_2[S,B]!\text{ans}\langle 1\rangle$	$s_1[S,B]!\text{ans}\langle 2\rangle$
	$s_2[B,S]?\text{req}(\varepsilon)$	$s_1[B,S]?\text{req}(\varepsilon)$	$s_2[S,B]!\text{ans}\langle 1\rangle$	$s_1[S,B]!\text{ans}\langle 2\rangle$
(I) states	$\mathbf{t}\mapsto 1, \mathbf{c}\mapsto\{1\}$	$\mathbf{t}\mapsto 2, \mathbf{c}\mapsto\{1,2\}$	$\mathbf{t}\mapsto 2, \mathbf{c}\mapsto\{2\}$	$\mathbf{t}\mapsto 2, \mathbf{c}\mapsto\{\}$
(II) actions	$s_1[B,S]?\text{req}(\varepsilon)$	$s_2[B,S]?\text{req}(\varepsilon)$	$s_1[S,B]!\text{ans}\langle 2\rangle$	$s_2[S,B]!\text{ans}\langle 1\rangle$
	$s_2[B,S]?\text{req}(\varepsilon)$	$s_1[B,S]?\text{req}(\varepsilon)$	$s_1[S,B]!\text{ans}\langle 2\rangle$	$s_2[S,B]!\text{ans}\langle 1\rangle$
	$s_1[B,S]?\text{req}(\varepsilon)$	$s_2[B,S]?\text{req}(\varepsilon)$	$s_2[S,B]!\text{ans}\langle 2\rangle$	$s_1[S,B]!\text{ans}\langle 1\rangle$
	$s_2[B,S]?\text{req}(\varepsilon)$	$s_1[B,S]?\text{req}(\varepsilon)$	$s_2[S,B]!\text{ans}\langle 2\rangle$	$s_1[S,B]!\text{ans}\langle 1\rangle$
(II) states	$\mathbf{t}\mapsto 1, \mathbf{c}\mapsto\{1\}$	$\mathbf{t}\mapsto 2, \mathbf{c}\mapsto\{1,2\}$	$\mathbf{t}\mapsto 2, \mathbf{c}\mapsto\{1\}$	$\mathbf{t}\mapsto 2, \mathbf{c}\mapsto\{\}$
(III) actions	$s_1[B,S]?\text{req}(\varepsilon)$	$s_1[S,B]!\text{ans}\langle 1\rangle$	$s_2[B,S]?\text{req}(\varepsilon)$	$s_2[S,B]!\text{ans}\langle 2\rangle$
	$s_2[B,S]?\text{req}(\varepsilon)$	$s_2[S,B]!\text{ans}\langle 1\rangle$	$s_1[B,S]?\text{req}(\varepsilon)$	$s_1[S,B]!\text{ans}\langle 2\rangle$
(III) states	$\mathbf{t}\mapsto 1, \mathbf{c}\mapsto\{1\}$	$\mathbf{t}\mapsto 1, \mathbf{c}\mapsto\{\}$	$\mathbf{t}\mapsto 2, \mathbf{c}\mapsto\{2\}$	$\mathbf{t}\mapsto 2, \mathbf{c}\mapsto\{\}$

Figure 8.1: The Valid Traces of Asynchronous and Synchronous Interactions w.r.t. G_{assign}

2. If S issues a number from this set, the remaining numbers are what it can issue next.

As we shall see later, the stateful protocol has the property that, if S behaves well w.r.t. G_{assign}, then a remote observer also finds it well-behaved.

To understand G_{assign} as a stateful protocol, consider two sessions following the protocol, s_1 and s_2. Assume the initial states are $\mathbf{t}\mapsto 0$ and $\mathbf{c}\mapsto\{\}$. Then G_{assign} says the traces in Figure 8.1 are valid ones. Figure 8.1 lists the traces together with step-by-step state change: (I,II,III) are categories each stipulating how states will change.

Above, $s_1[B,S]?\text{req}(\varepsilon)$ denotes an input ? from B to S at session s_1 carrying a req-message without value; $s_1[S,B]!\text{ans}\langle 1\rangle$ is an output ! from S to B at s_1 carrying a ans-message with value 1. (I) and (II) are the traces where a remote observer observes that two consecutive inputs have arrived first. Note that, even if S may have indeed outputted immediately after the first input, we can have these traces due to asynchrony. Even then, unlike G_{async}, the observer is always sure, by G_{assign}, that the returned values should be no more than 2, i.e. it is either 1 or 2. In (III), the observer observes the second request only after the answer to the first request. Note the request-answer order

in each session is preserved because without the request, its answer cannot occur (which is Lamport's ordering (98)). Note that, the order of two messages inside each session is preserved. Unlike G_{async}, the observer can expect, based on G_{assign}, that the first answer is surely 1, and the second is surely 2.

The above example shows how we can represent causality while still (intuitively) keeping the asynchronous nature of specifications. It also shows how traces of actions can be used to give extensional meaning of specifications. This observation is given a formal account in Section 8.3 later.

8.1.4 Analysis of Capturing Causality: G_{assign}

Continue with section 8.1.3, from G_{assign} we have learnt that an observer can be more insightful if she has more information (which are initially established by input actions).

Convention 8.1.4. An observer (remote/close) can expect and reasonably guess the behaviours of her observees based on the messages she has observed. Without loss of generality, we assume that the messages delivered by observees can represent the behaviours of observees.

Based on G_{assign}, in which the assigned number to each session (established at each request) is asked to be increased with the coming requests one by one, the following properties say that a remote observer can expect the next outputted value based on what she observes. Let a sequence of actions, called *trace*, denoted by $\mathtt{s}$. Let $\mathtt{I}(\mathtt{s})$ be the sequence of input actions of $\mathtt{s}$, $\mathtt{O}(\mathtt{s})$ be the sequence of output actions of $\mathtt{s}$, and $\mathtt{V}(\mathtt{s})$ be the sequence of values carried in $\mathtt{s}$. Let $\mathsf{num}(\mathtt{s})$ be the length of $\mathtt{s}$, and $\mathtt{prefix}(\mathtt{s})$ be the set of prefixes of $\mathtt{s}$. Moreover, let $\mathtt{v}(\ell)$ be the value carried by action ℓ, and $\mathsf{ini}(\mathbf{t})$ be the initial value of state $\mathbf{t} \in G_{\mathsf{assign}}$. Because in the following cases we only discuss one state of an endpoint, we write $\mathsf{ini}(\mathbf{t})$ simply as ini.

For every trace w.r.t role S (representing server) satisfying $G_{\mathsf{assign}} \upharpoonright S$, it should have the following properties:

$\mathfrak{tq}0$. binding conventions.

$\mathfrak{tq}1$. $\mathsf{num}(\mathtt{I}(\mathtt{s})) \geq \mathsf{num}(\mathtt{O}(\mathtt{s}))$.

$\mathfrak{tq}2$. $\forall \ell \in \mathtt{s} \ \cap \ \ell = s[S,B]!l\langle v\rangle \ \subset \ \exists\, \ell' \in \mathtt{s},\, \ell' = s[B,S]?l(v)$.

$\mathfrak{rq}3$. $\forall \ell, \ell' \in \mathtt{O}(\mathtt{s}),\ \ell \neq \ell'$, then $\mathtt{v}(\ell) \neq \mathtt{v}(\ell')$.

$\mathfrak{rq}4$. $\forall \ell \in \mathtt{O}(\mathtt{s}'), \mathtt{s}' \in \mathtt{prefix}(\mathtt{s}), \mathsf{ini} \leq \mathtt{v}(\ell) \leq \mathsf{num}(\mathtt{I}(\mathtt{s}')) + \mathsf{ini}$.

The first 3 properties ($\mathfrak{rq}0$, $\mathfrak{rq}1$ and $\mathfrak{rq}2$) ensures the trace satisfies the interaction structure defined in G_{assign}. In $\mathfrak{rq}1$, operation $\mathtt{I}(\mathtt{s})$ extracts the input actions from trace $\mathtt{s}$ as a new trace, and $\mathsf{num}(\mathtt{I}(\mathtt{s}))$ means the length of input actions in $\mathtt{s}$. Similarly, operation $\mathtt{O}(\mathtt{s})$ extracts the output actions from trace $\mathtt{s}$ as a new trace $\mathsf{num}(\mathtt{O}(\mathtt{s}))$ means the length of output actions in $\mathtt{s}$. Overall it states that the number of inputs (i.e. requests from the buyers) should be bigger than the number of outputs (i.e. assignments from the server). $\mathfrak{rq}2$ states that only when a request takes place, an assignment takes place. $\mathfrak{rq}3$ ensures that every assigned (outputted) value (i.e. $\ell \in \mathtt{O}, (\mathtt{v}(\ell))$ is distinct. The operation $\mathtt{v}(\ell)$ extracts the value carried by action ℓ. In $\mathfrak{rq}4$, ini is a constant, denoting the initial value of a particular state used for checking outputted values. Here it is the initial value of state $\mathbf{c}$ defined in G_{assign}. $\mathfrak{rq}4$ states that, an outputted value carried by an output action should be smaller than or equal to the number of inputs which positioning ahead of it. $\mathfrak{rq}3$ and $\mathfrak{rq}4$ together imply that, for a given trace $\mathtt{s} = \ell_1 ... \ell_n$, the next outputted value has a fixed range: This range can be measured by knowing how many requests from buyers have occurred and how many assignments to buyers have done.

For example, assume $\mathsf{ini} = 1$; when an observer observes a trace $s_1[B,S]?\mathrm{req}(\varepsilon) \cdot s_2[B,S]?\mathrm{req}(\varepsilon) \cdot s_3[B,S]?\mathrm{req}(\varepsilon)$, if the next action is $s_2[S,B]!\mathrm{ans}\langle v \rangle$, then $v \in \{1,2,3\}$; assume $v = 3$. After action $s_2[S,B]!\mathrm{ans}\langle 3 \rangle$, if the upcoming one is $s_1[S,B]!\mathrm{ans}\langle v' \rangle$, then $v' \in \{1,2\}$ because 3 has been assigned; assume $v' = 1$. When a new request comes, says $s_4[B,S]?\mathrm{req}(\varepsilon)$, currently the overall actions starting from the beginning is

$$s_1[B,S]?\mathrm{req}(\varepsilon) s_2[B,S]?\mathrm{req}(\varepsilon) s_3[B,S]?\mathrm{req}(\varepsilon) s_2[S,B]!\mathrm{ans}\langle 3 \rangle s_1[S,B]!\mathrm{ans}\langle 1 \rangle s_4[B,S]?\mathrm{req}(\varepsilon)$$

the next output action of it can be $s_n[S,B]!\mathrm{ans}\langle v'' \rangle$, $n \in \{3,4\}$ and $v'' \in \{2,4\}$.

These analyses are useful for verifying a trace step by step. For example, assume the following trace is an initial trace: $s_1[B,S]?\mathrm{req}(\varepsilon) \cdot s_1[S,B]!\mathrm{ans}\langle 2 \rangle \cdot s_2[B,S]?\mathrm{req}(\varepsilon)$ is not right immediately at $s_1[S,B]!\mathrm{ans}\langle 2 \rangle$ because the only possible value at the moment when it is assigned is 1.

$$
\begin{array}{rcl@{\qquad}rcl}
S & ::= & \mathsf{nat} \mid \mathsf{bool} \mid \mathsf{string} & A & ::= & \mathsf{true} \mid \mathsf{false} \mid e_1 = e_2 \mid e_1 > e_2 \\
 & & \mid\ S_1 \times S_2 \mid \mathsf{set}(S) \mid \mathsf{map}(S_1, S_2) & & & \mid e_1 \in e_2 \mid\ A_1 \cap A_2 \mid \neg A \\
e & ::= & x \mid v \mid \mathbf{f} \mid op(e_1, ..., e_n) & E & ::= & \varepsilon \mid\ E, \mathbf{f} := e
\end{array}
$$

$$
\begin{array}{rcll}
G & ::= & p \to q : \{l_i(x_i : S_i)\langle\!\langle A_i; E_i \rangle\!\rangle \langle\!\langle A_i'; E_i' \rangle\!\rangle . G_i\}_{i \in I} & \text{G-cm} \\
 & \mid & G_1 \mid G_2 & \text{G-par} \\
 & \mid & \mathsf{end} & \text{G-end}
\end{array}
$$

$$
\begin{array}{rcll}
T & ::= & p!\{l_i(x_i : S_i)\langle\!\langle A_i; E_i \rangle\!\rangle . T_i\}_{i \in I} & \text{L-sel} \\
 & \mid & p?\{l_i(x_i : S_i)\langle\!\langle A_i; E_i \rangle\!\rangle . T_i\}_{i \in I} & \text{L-bra} \\
 & \mid & \mathsf{end} & \text{L-end}
\end{array}
$$

Figure 8.2: The Grammar of Stateful Protocols

8.2 The Language of Stateful Specifications: SP

8.2.1 The Syntax of SP

Figure 8.2 summarises the grammar of global stateful protocols $(G, \ldots)$ for specifying the interaction structure of a session from global viewpoints, and local stateful protocols $(T, \ldots)$, projected from the global G, for specifying protocols for local endpoints. Note that the "state" used here means the "fields of data storages". For example, it can be a field **counter** in a server for counting the number of visiters, or a field **credit** in a bank user for indicating the user's current savings, etc. This syntax enriches (22) with local states update and rich operations on them: By adding simple data update E and data type set, we obtain a rich class of stateful specifications (which consist of local SPs).

A state variable, denoted by $\mathbf{f}$, consists of zero or more *field*s. A field gets read in a *predicate* A and gets read and written in an *update* E. We call $\langle\!\langle A; E \rangle\!\rangle$ *obligation*. We use updates instead of post-conditions for usability in runtime verification. $(S, \ldots)$ are sorts (data types), and $(e, \ldots)$ are expressions, where $op(e_1, ..., e_n)$ is the operation op on parameters $e_1, ..., e_n$. We use data types such as product $S_1 \times S_2$, set $\mathsf{set}(S)$ and (finite) function $\mathsf{map}(S_1, S_2)$. Sets and functions play important roles in asynchronous specifications. In expressions, x is a variable, v is a value, $\mathbf{f}$ is a (mutable) field. In E, $\mathbf{f} := e$ stipulates that when $e \downarrow v$, $\mathbf{f}$ is updated with value v, which makes the value of $\mathbf{f}$ be v, denoted by $\mathbf{f} \mapsto v$. The grammars of G and T are simplified for distilled

presentation. They are similar to the ones defined in Figures 6.1 and 6.3 in Chapter 6, where G specifies the global interactions and T specifies the *obligation* for every action coming from endpoints. The recursion of G and T can be omitted because, while a recursion is finite, we can unfold the finitely-recursive *obligations* of the body to several continuous obligations, each of which is defined by T. All results are preserved.

In G, $p \to q$ describes the communication (interaction) from sender p to receiver q, while $p!$ and $p?$ are endpoint actions for output (to p) and input (from p) respectively. In $l_i(x_i : S_i)$, l_i is the label for a branch, when l_j is chosen, the interaction variable is x_j, and S_j is its type. In G-cm, the first obligation $\langle\langle A; E\rangle\rangle$ is for the sender, indicating the sender should guarantee that its message x satisfies A and as a result E is done; the second obligation $\langle\langle A'; E'\rangle\rangle$ is for the receiver, indicating it can expect a message x to satisfy A' and as a result E' is done. Rule G-par particularly asks that $\mathit{role}(G_1) \cap \mathit{role}(G_2) = \emptyset$, which means that no role is shared by G_1 and G_2, where $\mathit{role}(G)$ denotes the set of roles in G. Rule L-sel is for sender's behaviour, while rule L-bra is for receiver's behaviour. Parallel composition specifies two interactions in parallel, while end denotes the end of interactions. As a notational convention, if both obligations for the sender and the receiver are trivial (i.e. the predicate is true and the update is ε) then they are omitted.

8.2.2 Consistency Principles and Well-formedness

Before introducing the conditions of well-formedness for *stateful* global specifications, the set of states of a predicate or an update is defined as follows.

Definition 8.2.1 (the set of states). $\mathit{field}(C)$ denotes the set of states (i.e. fields) occurring in C. C can be a predicate (i.e. A) or an update (i.e. E).

Assume $p \to q : \{l_i(x_i : S_i)\langle\langle A_i; E_i\rangle\rangle\langle\langle A_i'; E_i'\rangle\rangle.G_i\}_{i\in I}$ is inside a context, with possibly preceding interactions. The following conditions of well-formedness for *stateful* global SPs, based on (22), stipulate the consistency principles of global stateful protocols:

1. (a) $\forall i \in I$, $\mathit{field}(A_i') = \emptyset$; and (b) $\forall i \in I$, A_i implies A_i'.
2. (history sensitivity) A_i and E_i only refer to interaction variables which p, the sender, has sent or received before, as well as x_i; moreover, A_i and E_i only contain the states of the sender. Similarly, A_i' and E_i' are for the receiver. Moreover, A_i and E_i only contain the states of the sender p, while E_i' only contains the states of the receiver q.

3. (temporal satisfiability) At each step, and for any state, there is always a branch with label l_i and a value x_i that satisfy A_i (hence A_i' at each step).

Condition 1.(a) says that a predicate of a receiver is stateless. Two reasons for this requirement: First, if a receiving-side predicate relies on its own local state, then the sender may not be able to find a "proper" value to send; secondly, a sent message, which has been verified by a trusted observer (e.g. a system monitor), should not be judged invalid by the predicate at the receiver-side (or rejected by the receiver-side's system monitor), otherwise it is inconsistent because it implies that the judgement at the sender-side is incorrect. Condition 1.(b) says that, in every interaction, the predicate at sender always implies the predicate at the receiver: Together with 1.(a), it means that if the sender sends a message that satisfies the sender's predicate, then automatically the receiver's predicate is satisfied (the latter is however useful for the receiver to know what to expect).

Example 8.2.2 (the sender's predicate always implies the receiver's). Let state $\mathbf{f}_q$ belong to role q. Assume $\mathbf{f}_q = 5$. Consider the following simple global SP which violates condition 1.(a):

$$p \to q \quad : \quad (x : \mathsf{int})\langle\langle \mathsf{true}\ ; \varepsilon\rangle\rangle\langle\langle \mathbf{f}_q < x\ ;\ \mathbf{f}_q := x\rangle\rangle.$$

When session participants apply this global SP for communication through a session, say s, if session-role $s[p]$ sends a message with value less than 5, this action will be rejected and considered invalid by the receiver-side observer. However, it is a valid action according to the sender-side observer who knows nothing about the predicate at the receiver side. Condition 1.(a) prevents this situation because all system monitors should be considered as having the same governance ability: Whenever the sender-side monitor approves an action according to the protocol, the receiver-side monitor should also approve it because the monitor at the sender side is trusted.

For the similar reason, condition 1.(b), which implies 1.(a), ensures that whenever the sender-side observer approves an action, the receiver-side agrees with it. The following global SP violates condition 1.(b):

$$p \to q \quad : \quad (x : \mathsf{int})\langle\langle x > 10\ ; \varepsilon\rangle\rangle\langle\langle x > 20\ ;\ \varepsilon\rangle\rangle.$$

This SP is not reasonable because, if the sender sends a value of x in the range from 11

to 20, from the sender's viewpoint, she obeys the protocol $x > 10$; however, she will be rejected by the receiver because she does not know that the receiver asks a value more than 20.

Conditions 2 and 3 are based on the consistency principles, history sensitivity and temporal satisfiability, introduced in Section 6.3, which are based on (22). The following examples illustrate the conditions 2 and 3, respectively.

Example 8.2.3 (history sensitivity). Let state $\mathbf{f}_q$ belong to role q. The following global SP satisfies history sensitivity with the similar reasoning of Example 6.3.1 in Section 6.3,

$$\begin{array}{lcl} p \to q & : & (x : \mathsf{int})\langle\!\langle x > 10 \ ; \varepsilon \rangle\!\rangle \langle\!\langle x > 5 \ ; \ \mathbf{f}_q := x \rangle\!\rangle. \\ q \to r & : & (x : \mathsf{int}). \\ r \to q & : & (y : \mathsf{int})\langle\!\langle y > x \ ; \varepsilon \rangle\!\rangle \langle\!\langle y > x \ ; \ \mathbf{f}_q := y \rangle\!\rangle \end{array}$$

because every participant p, q and r knows the variables deciding the predicates.

However, since the state $\mathbf{f}_q$ is only known by role q, when the constraint of $\mathbf{f}_q$ is added to the predicate of p, it violates the principle of history sensitivity:

$$\begin{array}{lcl} p \to q & : & (x : \mathsf{int})\langle\!\langle x > \mathbf{f}_q \ ; \varepsilon \rangle\!\rangle \langle\!\langle x > \mathbf{f}_q \ ; \ \mathbf{f}_q := x \rangle\!\rangle. \\ q \to r & : & (x : \mathsf{int}). \\ r \to q & : & (y : \mathsf{int})\langle\!\langle y > x \ ; \varepsilon \rangle\!\rangle \langle\!\langle y > x \ ; \ \mathbf{f}_q := y \rangle\!\rangle \end{array}$$

Example 8.2.4 (temporal satisfiability). Let state $\mathbf{f}_q$ belong to role q. The global SP below respects temporal satisfiability with the similar reasoning of Example 6.3.2 introduced in Section 6.3:

$$\begin{array}{lcl} p \to q & : & \mathrm{len}(x : \mathsf{int})\langle\!\langle 49 > x > 5 \ ; \ \varepsilon \rangle\!\rangle \langle\!\langle 49 > x > 0 \ ; \ \mathbf{f}_q := x + 10 \rangle\!\rangle. \\ q \to r & : & \{\mathrm{walk}(y : \mathsf{int})\langle\!\langle x < y < 10 \ ; \ \varepsilon \rangle\!\rangle \langle\!\langle x < y < 30 \ ; \ \varepsilon \rangle\!\rangle, \\ & & \quad \mathrm{run}(y : \mathsf{int})\langle\!\langle x < y < \mathbf{f}_q \ ; \ \varepsilon \rangle\!\rangle \langle\!\langle x < y < 60 \ ; \ \varepsilon \rangle\!\rangle\} \end{array}$$

although the state $\mathbf{f}_q$ is involved in the predicate at branch "run", since there are always integers between x and $x + 10 = \mathbf{f}_q$ and the seller's predicate $\langle\!\langle x < y < \mathbf{f}_q \rangle\!\rangle$ always implies the receiver's predicate $\langle\!\langle x < y < 60 \rangle\!\rangle$ when x is confined between 5 to 49, and $\mathbf{f}_q$ is updated to $x + 10$.

However, when the update of $\mathbf{f}_q$ is changed to $\mathbf{f}_q := x$, the global SP violates temporal

satisfiability because there is no path to go forward when the value of x is more than 9.

$$\begin{array}{rcl} p \to q & : & \mathrm{len}(x : \mathsf{int})\langle\!\langle 49 > x > 5 \ ; \ \varepsilon \rangle\!\rangle \langle\!\langle 49 > x > 0 \ ; \ \mathbf{f}_q := x \rangle\!\rangle. \\ q \to r & : & \{\mathrm{walk}(y : \mathsf{int})\langle\!\langle x < y < 10 \ ; \ \varepsilon \rangle\!\rangle \langle\!\langle x < y < 30 \ ; \ \varepsilon \rangle\!\rangle, \\ & & \quad \mathrm{run}(y : \mathsf{int})\langle\!\langle x < y < \mathbf{f}_q \ ; \ \varepsilon \rangle\!\rangle \langle\!\langle x < y < 60 \ ; \ \varepsilon \rangle\!\rangle\} \end{array}$$

Note that in Example 8.2.4, in fact, for the update of $\mathbf{f}_q := x + k$, when $k \leq 2$, the global SP always violates temporal satisfiability (Condition 3); when $k > 12$, it always voilates Condition 1.(b).

Note that, all examples treated in this chapter are well-formed. *Henceforth we assume all global SPs we treat are well-formed.*

8.2.3 The Projection from a Global SP to Local SPs

Global SPs are useful to capture overall interaction scenarios, while local SPs specify exactly what endpoints should do. They are linked by *endpoint projection*.

The projection rule defined in Definition 8.2.5 is similar to the one defined in Definition 6.5.1 in Chapter 6. Assume $p, q, r \in G$ and $p \neq q$. $var(G) \upharpoonright p$ is the set of variables that p knows in G.

Definition 8.2.5. Assume G is well-formed. The *stateful* projection $\mathrm{Proj}(G,\ A,\ r)$ is inductively defined as:

$$\begin{array}{l} \mathrm{Proj}(p \to q : \{l_i(x_i : S_i)\langle\!\langle A_i \ ; \ E_i \rangle\!\rangle \langle\!\langle A_i' \ ; \ E_i' \rangle\!\rangle . G_i\}_{i \in I},\ A^{pj},\ r) = \\ \quad \begin{cases} q!\{l_i(x_i : S_i)\langle\!\langle A_i \ ; \ E_i \ \rangle\!\rangle . G_i^{pj}\}_{i \in I} & \text{if } r = p \neq q, \\ p?\{l_i(x_i : S_i)\langle\!\langle A_i' \ ; \ E_i' \rangle\!\rangle . G_i^{pj}\}_{i \in I} & \text{if } r = q \neq p, \\ \sqcup_{i \in I} G_i^{pj} & \text{if } r \neq q \neq p \end{cases} \\ \quad \text{with } G_i^{pj} = \mathrm{Proj}(G_i, A^{pj}, r) \end{array}$$

$$\mathrm{Proj}(G_1 \mid G_2,\ A^{pj},\ r) = \begin{cases} \mathrm{Proj}(G_i,\ A^{pj}, r) & \text{if } r \in G_i \text{ and } r \notin G_j \ i, j \in \{1, 2\}, \\ \mathsf{end} & \text{else} \end{cases}$$

$$\mathsf{end} \upharpoonright r = \mathsf{end}$$

where $\sqcup_{i \in I} G_i^{pj} = G_1^{pj} \sqcup G_2^{pj} ... \sqcup G_n^{pj}$ if $I = \{1, ..., n\}$. The mergeability, $T_1 \sqcup T_2$, is defined in the end of Definition 6.5.1 in Chapter 6. The application of projection is illustrated by the following example.

Example 8.2.6 (endpoint projection). The local SPs projected from G_{assign} on role S and B respectively are:

$$\begin{array}{rcccl} G_{\mathsf{assign}} \upharpoonright S & = & T_S & = & B?\mathrm{req}(\varepsilon)\langle\!\langle \mathsf{true}; \mathbf{t} := \mathbf{t} + \mathbf{1} \;\; \mathbf{c} := \mathbf{c} \cup \{\mathbf{t}\}\rangle\!\rangle. \\ & & & & B!\mathrm{ans}(x : \mathsf{int})\langle\!\langle x \in \mathbf{c} \;;\; \mathbf{c} := \mathbf{c} \setminus \{x\}\rangle\!\rangle.\mathsf{end} \\ G_{\mathsf{assign}} \upharpoonright B & = & T_B & = & S!\mathrm{req}(\varepsilon).S?\mathrm{ans}(x : \mathsf{int}).\mathsf{end} \end{array}$$

8.3 Specifications Composed by SP

8.3.1 The Syntax of Stateful Specifications

A *specification* is a triple $\Theta ::= \langle \Gamma; \Delta; D \rangle$ which gives a behavioural specification of a local process (endpoint) as its interface. Γ, Δ and D, separated by ";" in Θ, are given by:

$$\begin{array}{rcl} \Gamma & ::= & \emptyset \mid \Gamma, a : \mathtt{mode}(G[p]) \mid \Gamma, \mathbf{f} : S \\ \Delta & ::= & \emptyset \mid \Delta, s[p] : T \\ D & ::= & \emptyset \mid D, \mathbf{f} \mapsto v \end{array}$$

We here simplify the modes of shared names as $\mathtt{mode} \in \{\mathtt{I}, \mathtt{O}\}$. Γ, *shared environment*, describes several information: The permitted behaviour at each shared channel; and the type of each field. When a process has $a : \mathtt{I}(G[p])$, it can *accept* invitations via shared channel a to play role p following what G specifies; while $a : \mathtt{O}(G[p])$ is its dual. In Δ, *session environment*, $s[p] : T$ describes the session behaviour (T) in a session s as p. D is a set of (ghost) states of a local process (endpoint): The states in $D \in \Theta$ belong to an endpoint participant in a session. Each D is a map from fields to values, storing a range of data structure. In formulae, a field $\mathbf{f}$ itself represents its current value.

Example 8.3.1. Based on G_{assign} in Example 8.1.3 and its local SPs in Example 8.2.6, we give a local specification Θ_{ass} for server, playing role S, and Θ_{B_1} and Θ_{B_2} for two buyers B_1 and B_2, each playing role B in G_{assign}, assuming there are two ongoing

sessions s_1 and s_2.

$$\begin{array}{rcl} T_{\mathsf{ass}} & = & B?\mathrm{req}(\varepsilon)\langle\!\langle \mathsf{true}; \mathbf{t} := \mathbf{t} + \mathbf{1} \;\; \mathbf{c} := \mathbf{c} \cup \{\mathbf{t}\}\rangle\!\rangle.T'_{\mathsf{ass}}, \\ T'_{\mathsf{ass}} & = & B!\mathrm{ans}(x : \mathsf{int})\langle\!\langle x \in \mathbf{c} \;;\; \mathbf{c} := \mathbf{c} \setminus \{x\}\rangle\!\rangle.\mathsf{end} \\ \Theta_{\mathsf{ass}} & = & \langle \Gamma'_{\mathsf{ass}}, \mathtt{server} : \mathtt{I}(G_{\mathsf{ass}}[S]);\; \Delta'_{\mathsf{ass}}, s_1[S] : T_{\mathsf{ass}}, s_2[S] : T_{\mathsf{ass}};\; D'_{\mathsf{ass}}, \mathbf{t} \mapsto 0, \mathbf{c} \mapsto \{\} \rangle \end{array}$$

$$\begin{array}{rcl} T_B & = & S!\mathrm{req}(\varepsilon).T'_B, \\ T'_B & = & S?\mathrm{ans}(x : \mathsf{int}).\mathsf{end}, \\ \Theta_{B_1} & = & \langle \Gamma'_{B_1}, \mathtt{b}_1 : \mathtt{I}(G_B[B]), \mathtt{server} : \mathtt{O}(G_{\mathsf{ass}}[S]) \;;\; \Delta'_{B_1}, s_1[B] : T_B;\; D_{B_1} \rangle \\ \Theta_{B_2} & = & \langle \Gamma'_{B_2}, \mathtt{b}_2 : \mathtt{I}(G_B[B]), \mathtt{server} : \mathtt{O}(G_{\mathsf{ass}}[S]) \;;\; \Delta'_{B_2}, s_2[B] : T_B;\; D_{B_2} \rangle \end{array}$$

The data storage in Θ_{ass} is $D'_{\mathsf{ass}}, \mathbf{t}, \mathbf{c}$. In this protocol, no state in D'_{ass} is used. Similarly, no state in D_{B_1} or D_{B_2} is used. Although we do not illustrate the whole procedures of session establishment (by using rules [Acc] and [Req] defined in Figure 8.3), it shows that buyers B_1 and B_2 are the session inviters requesting S to join session s_1 and s_2.

8.3.2 The Semantics of Stateful Specifications

Figure 8.3 represents the semantics of specifications as a deterministic labelled transition system, of the form $\Theta \xrightarrow{\ell} \Theta'$, which intuitively means Θ as a specification *allows* a process to do an action ℓ, and demands the resulting process to conform to Θ'.

Definition 8.3.2.

$$\Theta \xrightarrow{\ell_1} \Theta_1 \xrightarrow{\ell_2} \Theta_2 .... \xrightarrow{\ell_k} \Theta_k ... \stackrel{\text{def}}{=} \Theta \xrightarrow{\ell_1\ell_2...\ell_k} \Theta_k ...$$

For actions labels, here we adopt the simplified version of those defined in Equation 5.1. Because this chapter, for specifying asynchronous specification when endpoints' states are also stipulated, focuses on the *session interactions*, actions $\mathsf{new}\ a : \mathtt{im}(G[p])$ and $\mathtt{join}(s[p])$, which are control actions, are not discussed in the LTS defined in Figure 8.3. The rules of bound/free request and bound/free accept of the semanics are similar to those of the LTS of monitors defined in Figure 7.4, Chapter 7. We do not use τ since it is irrelevant in the present work (because, in brief, τ is always possible and has no effects on specifications). The special rules for stateful specifications are rules [Sel/SelN] and [Bra/BraN]. Rule [Sel] is for sending a message inside a session. The premise says that, first, the type T should be a selection type; the passed value v has type S_j from the jth branch of T under Γ (note that, when v is a name, Γ is necessary

Actions $\quad \ell ::= \tau \mid \overline{a}(s[p] : G) \mid \overline{a}\langle s[p] : G\rangle \mid a(s[p] : G) \mid a\langle s[p] : G\rangle \mid s[p,q]!l\langle v\rangle \mid s[p,q]?l(v)$

[Req-ini] $$\frac{s \notin \mathsf{dom}(\Delta),\ \forall i \in I, p_i \in \mathit{role}(G),\ G \text{ is well-formed},\ \forall k,j \in I, p_k \neq p_j\ (k \neq j)}{\langle \Gamma, a : \mathtt{0}(G[p_j]);\ \Delta;\ D\rangle \xrightarrow{\overline{a}(s[p_j]:G)} \langle \Gamma, a : \mathtt{0}(G[p_j]);\ \Delta, \{s[p_i] : G \upharpoonright p_i\}_{i \in I \setminus \{j\}};\ D\rangle}$$

[Req] $$\frac{p_j \in \mathsf{role}(G)}{\langle \Gamma, a : \mathtt{0}(G[p_j]);\ \Delta, s[p_j] : G \upharpoonright p_j;\ D\rangle \xrightarrow{\overline{a}\langle s[p_j]:G\rangle} \langle \Gamma, a : \mathtt{0}(G[p_j]);\ \Delta;\ D\rangle}$$

[Acc-b] $$\frac{s \notin \mathsf{dom}(\Delta),\ T = G \upharpoonright q,\ \mathit{field}(T) \in D}{\langle \Gamma, a : \mathtt{I}(G[q]);\ \Delta;\ D\rangle \xrightarrow{a(s[q]:G)} \langle \Gamma, a : \mathtt{I}(G[q]);\ \Delta, s[q] : T;\ D\rangle}$$

[Acc-f] $$\frac{s \in \mathsf{dom}(\Delta),\ T = G \upharpoonright q,\ \mathit{field}(T) \in D}{\langle \Gamma, a : \mathtt{I}(G[q]);\ \Delta;\ D\rangle \xrightarrow{a\langle s[q]:G\rangle} \langle \Gamma, a : \mathtt{I}(G[q]);\ \Delta, s[q] : T;\ D\rangle}$$

[Sel] $$\frac{T \curvearrowright q!\{l_i(x_i : S_i)\langle\!\langle A_i; E_i\rangle\!\rangle.T_i\}_{i \in I},\ \ \Gamma \vdash v : S_j,\ \ \Gamma \models A_j\{v/x_j\},}{\langle \Gamma;\ \Delta, s[p] : T;\ D\rangle \xrightarrow{s[p,q]!l_j\langle v\rangle} \langle \Gamma;\ \Delta, s[p] : T_j\{v/x_j\};\ D \,\mathtt{after}\, E_j\{v/x_j\}\rangle}$$

[SelN] $$\frac{T \curvearrowright q!\{l_i(x_i : S_i)\langle\!\langle A_i; E_i\rangle\!\rangle.T_i\}_{i \in I},\ \Gamma \vdash a : S_j,\ a \notin (A_j \cap E_j),\ \ A_j \downarrow \mathsf{true},}{\langle \Gamma;\ \Delta, s[p] : T;\ D\rangle \xrightarrow{s[p,q]!l_j\langle a\rangle} \langle \Gamma;\ \Delta, s[p] : T_j;\ D \,\mathtt{after}\, E_j\rangle}$$

[Bra] $$\frac{T \curvearrowright p?\{l_i(x_i : S_i)\langle\!\langle A_i; E_i\rangle\!\rangle.T_i\}_{i \in I},\ \ \Gamma \vdash v : S_j,\ \ \Gamma \models A_j\{v/x_j\},}{\langle \Gamma;\ \Delta, s[q] : T;\ D\rangle \xrightarrow{s[p,q]?l_j(v)} \langle \Gamma;\ \Delta, s[q] : T_j\{v/x_j\};\ D \,\mathtt{after}\, E_j\{v/x_j\}\rangle}$$

[BraN] $$\frac{T \curvearrowright p?\{l_i(x_i : S_i)\langle\!\langle A_i; E_i\rangle\!\rangle.T_i\}_{i \in I},\ S_j = \mathtt{mode}(G'[p']),\ a \notin (A_j \cap E_j),\ A_j \downarrow \mathsf{true}}{\langle \Gamma;\ \Delta, s[q] : T;\ D\rangle \xrightarrow{s[p,q]?l_j(a)} \langle \Gamma, a : \mathtt{0}(G'[p']);\ \Delta, s[q] : T_j;\ D \,\mathtt{after}\, E_j\rangle}$$

[Par] $$\frac{\Theta_1 \xrightarrow{\ell} \Theta_2,}{\Theta_1, \Theta_3 \xrightarrow{\ell} \Theta_2, \Theta_3}$$

Figure 8.3: The Labelled Transition System of Specifications

to have the knowledge of its type, but it is not needed if v is a non-name value, like 3 or "*hello*" because its type is automatically known without Γ); and A_j after substitution is well-typed under Γ. In the conclusion, T'_j substitutes v for x_j and prepares for the next (incoming or outgoing) message, and the state is updated by $D\,\texttt{after}\,E_j\{v/x_j\}$. To illustrate the updating of D by E_j, assume E_j is defined as $\mathbf{f} := \mathbf{f} \cup \{x_j\}$, and currently $\mathbf{f} \mapsto \{10\}$. After substituting 5 for x_j, D is updated to $\mathbf{f} \mapsto \{10, 5\}$. Rule [SelN] is similar to rule [Sel] except that it is particularly for sending a name, which should not be in A_j and A_j should be evaluated as true. Rule [Bra] is a symmetric rule of [Sel]. Rule [BraN] is for receiving shared channel a. It is dual to rule [SelN], and similar to the rule defined in 7.4.

We define two specifications, say Θ and Θ', are composable whenever $\mathsf{P}(\Theta) \cap \mathsf{P}(\Theta') = \emptyset$, and write Θ, Θ' as a composition of specification Θ and specification Θ. It is defined similarly as Definition 7.4.23, which defines the parallel composition of monitors. [Par] says, If Θ_1 and Θ_3 are composable, after action happens and Θ_1 becomes as Θ_2, they are still composable.

8.4 Traces of Local Actions

In this section, the trace of a local process is defined to describe a local process's run-time behaviour, and define the satisfaction criterion for formally describing a process satisfies a given specification synchronously/asynchronously. Note that, hereafter, when we say a trace, it always means a trace consisting of local actions inputting or outputting from a local process.

Definition 8.4.1 (trace). A *trace* $(\mathtt{s}, \mathtt{s}', \ldots)$ is a sequence of actions represented by

$$\mathtt{s} = \ell_1 \cdot \ell_2 \ldots \cdot \ell_n \ldots$$

in which each action is connected by " $\cdot$ ".

Note that, for every $\mathtt{s}$, we assume an accept/request action introducing a session, say s, binds the later occurrences of s.

Definition 8.4.2 (valid trace according to Θ). A trace $\mathtt{s}$ is valid according to Θ if there always exists Θ' such that $\Theta \xrightarrow{\mathtt{s}} \Theta'$.

8.4.1 Permutation of Stateful Protocols

We formalise the notion of minimum permutations for local Θ, which is similar to the permutation mechanism defined in Definition 6.6.8 in Chapter 6. Having permutation rules for Θ (see Definition 8.4.4) makes an observer (i.e. system monitor), who observes actions based on Θ, be able to capture the causality relation of actions when the order of actions is not preserved due to asynchrony. For example, in one session, when an endpoint, say $s[p]$, waits for two input actions from different senders, say $s[q_1]$ and $s[q_2]$, the local Θ at role $s[p]$ is

$$q_1?l_1(x_1)\langle\langle A_1; E_1\rangle\rangle \cdot q_2?l_2(x_2)\langle\langle A_2; E_2\rangle\rangle$$

then the order of actions may be $s[q_1, p]?l_1(v_1) \cdot s[q_2, p]?l_2(v_2)$ or $s[q_2, p]?l_1(v_2) \cdot s[q_1, p]?l_1(v_1)$ because there is no way to restrict that the first action should firstly happen and arrive earlier at $s[p]$ than the second one.

The permutation rules for Θ have different purpose from the definition of commutativity (see Definition 8.5.31). The permutation rules for Θ are the *mechanisms* (or tools) that let an observer use them to properly observe actions, whose order is not preserved; while the attribute of commutativity of a Θ is used to validate whether the Θ is asynchronous verifiable or not.

Here we define stateful protocol context and local unit permutation. Assume that, whenever we write $\mathcal{T}[T]$, it always comes from a well-formed global stateful protocol (i.e. G). Recall that the well-formedness principles of global stateful protocol is introduced in section 8.2.2. If it is a local Θ containing the local SP projected from a non-well-formed global SP, it is not meaningful to explore the permutation mechanism when the original global stateful protocol is in a non-reasonable shape. For convenience, let an obligation $\langle\langle A; E\rangle\rangle$ for sender be $\langle\langle \eta_! \rangle\rangle$, and an obligation $\langle\langle A'; E'\rangle\rangle$ for receiver be $\langle\langle \eta_? \rangle\rangle$.

Definition 8.4.3 (Local SP context). Let $\dagger ::= \{?, !\}$. A *local stateful-session type context* $\mathcal{T}[\ ]$ is a local stateful protocol with a hole, whose grammar is defined as:

$$\mathcal{T}[\ ] \quad ::= \quad [\ ] \mid p\dagger : \{l_1(x_1 : S_1)\langle\langle \eta_{\dagger 1} \rangle\rangle.T_1, .., l_k(x_k : S_k)\langle\langle \eta_{\dagger k} \rangle\rangle.\mathcal{T}[\ ], .., l_n(x_n : S_n)\langle\langle \eta_{\dagger n} \rangle\rangle.T_n\}$$

$\mathcal{T}[T]$ denotes the result of filling the hole in the local SP context $\mathcal{T}$ with local SP T. Note that $[\ ]$ defines the identity function, i.e. $[T] = T$.

The causal order occurs when there are two consecutive sending (or receiving) actions with the same target (e.g. $q!l(v).q!l'(v')$ or $p?l(x).p?l'(x')$), or there is an input action followed by an output action.

Definition 8.4.4 (local unit permutation). The *unit permutation* of local SP is defined by the axioms below, assuming $p_1 \neq p_2$:

$$\begin{aligned}
&\mathcal{T}[p_1!\{l_i(x_i : S_i)\langle\langle \eta_{!i}\rangle\rangle.\underline{p_2!}\{l'_j(x'_j : S'_j)\langle\langle \eta'_{!j}\rangle\rangle.T_{ij}\}_{j\in I}\}_{i\in I}] \\
&\qquad\qquad \curvearrowright^1 \quad \mathcal{T}[\underline{p_2!}\{l'_j(x'_j : S'_j)\langle\langle \eta'_{!j}\rangle\rangle.p_1!\{l_i(x_i : S_i)\langle\langle \eta_{!i}\rangle\rangle.T_{ij}\}_{i\in I}\}_{j\in I}] \\
&\mathcal{T}[p_1?\{l_i(x_i : S_i)\langle\langle \eta_{?i}\rangle\rangle.\underline{p_2?}\{l'_j(x'_j : S'_j)\langle\langle \eta'_{?j}\rangle\rangle.T_{ij}\}_{j\in I}\}_{i\in I}] \\
&\qquad\qquad \curvearrowright^1 \quad \mathcal{T}[\underline{p_2?}\{l'_j(x'_j : S'_j)\langle\langle \eta'_{?j}\rangle\rangle.p_1?\{l_i(x_i : S_i)\langle\langle \eta_{?i}\rangle\rangle.T_{ij}\}_{i\in I}\}_{j\in I}] \\
&\mathcal{T}[p_1!\{l_i(x_i : S_i)\langle\langle \eta_{!i}\rangle\rangle.\underline{p_2?}\{l'_j(x'_j : S'_j)\langle\langle \eta'_{?j}\rangle\rangle.T_{ij}\}_{j\in I}\}_{i\in I}] \\
&\qquad\qquad \curvearrowright^1 \quad \mathcal{T}[\underline{p_2?}\{l'_j(x'_j : S'_j)\langle\langle \eta'_{?j}\rangle\rangle.p_1!\{l_i(x_i : S_i)\langle\langle \eta_{!i}\rangle\rangle.T_ij\}_{i\in I}\}_{j\in I}]
\end{aligned}$$

8.4.2 Permutation of Local Actions

Since Θ is a specification for a local endpoint, the actions concerned here are all *local* actions, which means that they are the actions happening at endpoints. Similarly, all permutations of actions are the permutation of local actions. Example below shows the relations between global actions and local actions.

Example 8.4.5 (global actions v.s. local actions). Assume a session s is given, in which the roles involved in it are p_1 and p_2. Assume globally, there are actions among session-roles $s[p_1]$ and $s[p_2]$:

$$\begin{aligned}
&s[p_1, p_2]!\langle\text{"How are you?"}\rangle \cdot s[p_1, p_2]?(\text{"How are you?"}) \cdot \\
&s[p_2, p_1]!\langle\text{"Fine. Thanks!"}\rangle \cdot s[p_2, p_1]?(\text{"Fine. Thanks!"})
\end{aligned}$$

with the scenario: An endpoint playing role p_1 in session s, denoted by $s[p_1]$, firstly sends a regard "How are you" to an endpoint playing role p_2 in session s, denoted by $s[p_2]$. After $s[p_2]$ receives this regards, she replies $s[p_1]$ with "Fine. Thanks!"; then $s[p_1]$ receives this response. The sequence of actions can be observed by the following messages

$$s\langle p_1, p_2, \langle\text{"How are you?"}\rangle\rangle \cdot s\langle p_2, p_1, \langle\text{"Fine. Thanks!"}\rangle\rangle$$

This sequence of messages reflect the meaning of the sequence of actions.

Locally, the sequence of actions for session-role $s[p_1]$ is

$$s[p_1, p_2]!\langle\text{"How are you?"}\rangle \cdot s[p_2, p_1]?(\text{"Fine. Thanks!"})$$

while the sequence of actions for session-role $s[p_2]$ is

$$s[p_1, p_2]?(\text{"How are you?"}) \cdot s[p_2, p_1]!\langle\text{"Fine. Thanks!"}\rangle$$

A remote observer generally has more than one specifications (for different session-roles). For each session-role, the specification only specifies the local behaviours of that session-role.

To capture the causality among actions, below we define the relation of *dependency*:

Definition 8.4.6 (dependency). We define $\mathfrak{D}$, the dependency relation, such that $\ell_1 \mathfrak{D} \ell_2$ whenever

(a) ℓ_1 is either $a(k[p] : G)$ or $a\langle k[p] : G\rangle$, and $k[p] \in \ell_2$ or $\mathsf{sbj}(\ell_2) = k[p]$.

(b) ℓ_1 is $s[p_1, p_2]?l(x)$ and $x \in \ell_2$.

We say that ℓ_2 depends on ℓ_1 if $\ell_1 \mathfrak{D} \ell_2$.

The following examples illustrate the cases of dependency relation over action labels.

Example 8.4.7. Assume a sequence of actions $\ell_1 \cdot \ell_2$, where

$$\ell_1 = a\langle k[p] : G\rangle,\ \ell_2 = \overline{b}\langle k[p] : G\rangle.$$

Based on Definition 8.4.6.(a), ℓ_2 depends on ℓ_1.

Example 8.4.8. Assume a sequence of local actions of some local endpoint $\mathsf{s} = \mathsf{s}_0 \cdot \ell_1 \cdot \ell_2$, where

$$\ell_1 = a(k[p] : G),\ \ell_2 = k[p', p]!l\langle v\rangle$$

ℓ_2 does not depend on ℓ_1 because, based on Definition 8.4.6.(a), $\mathsf{sbj}(\ell_2) = k[p'] \neq k[p]$.

Example 8.4.9. Assume a sequence of local actions of some local endpoint $\mathsf{s} = \mathsf{s}_0 \cdot \ell_1 \cdot \ell_2 \cdot \ell_3 \cdot \ell_4$, where

$$\ell_1 = a(k[p] : G),\ \ell_2 = k[p, q]!l\langle v\rangle,\ \ell_3 = b\langle k'[q] : G'\rangle,\ \ell_4 = k'[p, q]?l'(v')$$

ℓ_2 depends on ℓ_1 because, based on Definition 8.4.6.(a), $\mathsf{sbj}(\ell_2) = k[p]$. Similarly, ℓ_4 depends on ℓ_3.

Definition 8.4.10 (suppress of local actions). For a sequence of local actions $\ell_1 \cdot \ell_2$ of an endpoint, we say ℓ_2 is suppressed by ℓ_1 if

1. $\ell_1 = s[p_1, q]?l(v)$ and $\ell_2 = s[q, p_2]!l'\langle v' \rangle$, or
2. $\ell_1 = s[p, q]!l\langle v \rangle$ and $\ell_2 = s[p, q]!l'\langle v' \rangle$, or
3. $\ell_1 = s[p, q]?l(v)$ and $\ell_1 = s[p, q]?l'(v')$.

Definition 8.4.11 (causality). We say local actions ℓ_1 and ℓ_2 of an endpoint have causality relation if one of them depends or suppresses the other.

Definition 8.4.12 (legal unit permutation of local actions). Let sequence of local actions $\ell_1 \cdot \ell_2$ be a trace of an endpoint. A permutation from $\ell_1 \cdot \ell_2$ to $\ell_2 \cdot \ell_1$ is *legal* if ℓ_1 and ℓ_2 have no causality relation.

We write $\mathtt{s} \curvearrowright \mathtt{s}'$ when $\mathtt{s}'$ is the result of applying zero or *more* legal unit permutations. In this case $\mathtt{s}'$ is *a permutation variant of* $\mathtt{s}$ and this permutation is called a *legal permutation*.

Definition 8.4.13 (valid unit permutation of actions according to Θ). We say $\mathtt{s}_1$ *is a permutation variant of* $\mathtt{s}_2$ *under* Θ, written $\Theta \vdash \mathtt{s}_1 \curvearrowright \mathtt{s}_2$, when $\mathtt{s}_1 \curvearrowright \mathtt{s}_2$, $\Theta \vdash \mathtt{s}_1$ and $\Theta \vdash \mathtt{s}_2$ hold.

Example 8.4.14 (legal permutations of actions). In Figure 8.1, all traces in (I) and (II) are permutation variants to each other. The traces in (III) can legally permute to any trace in (I) and (II), but not the converse.

Note that, it does not mean that a sequence of legally-permuted actions is always valid to a Θ. The following example explains this point.

Example 8.4.15. Let $\ell_1 = s[p, q_1]!\mathrm{ans}\langle\varepsilon\rangle$ and $\ell_2 = s[p, q_2]!\mathrm{ans}\langle\varepsilon\rangle$. Then $\ell_1 \cdot \ell_2 \curvearrowright \ell_2 \cdot \ell_1$. Assume that initial value of state $\mathbf{c}$ is 1, and that local SP for role p in session s is:

$$s[p] : q_1!\mathrm{ans}(\varepsilon)\langle\!\langle \mathsf{true}; \varepsilon \rangle\!\rangle \ . \ q_2!\mathrm{ans}(\varepsilon)\langle\!\langle \mathbf{c} = 1; \mathbf{c} := 0 \rangle\!\rangle$$

Then $\Theta \vdash \ell_1 \cdot \ell_2$, but $\Theta \nvdash \ell_2 \cdot \ell_1$.

8.5 Theory for Stateful Specifications

8.5.1 Satisfaction

The following simple definition of processes is enough for our purpose: We can readily use the π-calculus with session primitives and its weak (τ-abstracted) LTS to induce this abstract notion of processes.

Definition 8.5.1 (process). A *process* $(P, Q, ..)$ is a prefix-closed set of traces.

The following defines the notion of synchronous and asynchronous observables as the set of traces observed by a synchronous observer (i.e. as it is) and by an asynchronous observer (i.e. up to legal permutations).

Definition 8.5.2 (synchronous and asynchronous observable).

1. $\mathsf{Obs}_s(P) \stackrel{\text{def}}{=} P$.
2. $\mathsf{Obs}_a(P)$ is the set of all legal permutation variants of the traces in P.

Firstly, recall the LTS rule of SP defined in Figure 8.3: a valid ℓ against a Θ is an action which can be verified by a LTS rule such that $\Theta \xrightarrow{\ell} \Theta'$. Intuitively, a valid trace is a trace that Θ approves it.

Definition 8.5.3 ($\mathtt{trace}(\Theta)$: the set of valid traces of Θ). Define $\mathtt{trace}(\Theta)$ as the set of *valid traces* of Θ:

$$\mathtt{trace}(\Theta) = \{\mathtt{s} \mid \Theta \xrightarrow{\mathtt{s}} \Theta'\}$$

i.e. $\mathtt{trace}(\Theta)$ is the maximun set of traces such that $\forall \mathtt{s} \in \mathtt{trace}(\Theta), \Theta \vdash \mathtt{s}$.

Definition 8.5.4 (satisfaction up to observables). A process $\mathsf{Obs}_s(P)$ *synchronously satisfies* Θ, denoted $P \models_{\mathtt{sync}} \Theta$, when the following two conditions hold:

1. (output safety) $\mathsf{Obs}_s(P) \subset \mathtt{trace}(\Theta)$.

2.a (input consistency) Whenever $\mathtt{s} \in \mathsf{Obs}_s(P)$ and $\mathtt{s} \cdot \ell \in \mathtt{trace}(\Theta)$ where $\ell = s[p, q]?l(v)$ is an input, $\mathtt{s} \cdot \ell' \in \mathsf{Obs}_s(P)$ and ℓ' is an input with the shape $\ell' = s[p, q]?l'(v')$, then $\mathtt{s} \cdot \ell \in \mathsf{Obs}_s(P)$.

A process P *asynchronously satisfies* Θ, denoted $P \models_{\mathtt{async}} \Theta$, if, after replacing each $\mathsf{Obs}_s(P)$ with $\mathsf{Obs}_a(P)$, it satisfies condition 1. above and the following condition:

2.b (input consistency) Whenever $\mathtt{s} \in \mathsf{Obs}_a(P)$ and $\mathtt{s} \cdot \ell \in \mathtt{trace}(\Theta)$ where ℓ is an input, then $\mathtt{s} \cdot \ell \in \mathsf{Obs}_a(P)$.

Note that, for synchronous process (2.a), it can accept a valid input only when it is ready to receive it; while for asynchronous process (2.b), it can accept and should accept a valid input. Intuitively, Definition 8.5.4 says that a process P synchronously (resp. asynchronously) satisfies Θ if, w.r.t. synchronous (resp. asynchronous) observables, P always does valid outputs as far as it receives valid inputs.

Example 8.5.5 (valid/invalid traces according to G_{assign}). We consider Θ_{ass} from Example 8.3.1 which uses the local SP, projected from G_{assign} (given in Section 8.1.3), for the server side. Then, for example,

$$s_2[B,S]?\mathrm{req}(\varepsilon) \cdot s_2[S,B]!\mathrm{ans}\langle 1\rangle \cdot s_1[B,S]?\mathrm{req}(\varepsilon) \cdot s_1[S,B]!\mathrm{ans}\langle 2\rangle$$

is a valid trace of Θ_{ass}, but

$$s_2[B,S]?\mathrm{req}(\varepsilon) \cdot s_2[S,B]!\mathrm{ans}\langle 2\rangle \cdot s_1[B,S]?\mathrm{req}(\varepsilon) \cdot s_1[B,S]!\mathrm{ans}\langle 1\rangle$$

is not its trace (violation is at the second step), because it is not permitted by a remote observer with Θ_{ass} when she observes it.

Lemma 8.5.6. $\mathtt{s} \in \mathsf{Obs}_s(P), P \models_{\mathtt{sync}} \Theta$ *implies* $\mathtt{s} \in \mathit{trace}(\Theta)$.

Proof. Directly from Definition 8.5.4. ■

8.5.2 Asynchronously Verifiable Specifications

We say Θ is *asynchronous* if it is suitable for a remote observer to verify the behaviour of a process. In this case, we do not want the conformance of a trace to change depending on an accidental reordering due to asynchrony, i.e. we want its validity to be robust w.r.t. legal permutations. In the follows, the protocols introduced in Section 8.1.2 are used.

Definition 8.5.7 (asynchronously verifiable specification). We say Θ is *asynchronously verifiable* or simply *asynchronous* when $\mathtt{s} \in \mathtt{trace}(\Theta)$ and $\mathtt{s} \curvearrowright \mathtt{s}'$ imply $\mathtt{s}' \in \mathtt{trace}(\Theta)$.

To check violation of asynchrony of a specification, we only have to find a single acceptable trace whose permutation is not acceptable.

Example 8.5.8. This example uses G_{sync} introduced in Example 8.1.2, Section 8.1.2. Let T_{sync} be the local SP at server, projected from G_{sync}.

$$\Theta_{\mathsf{sync}} = \langle \Gamma'_{\mathsf{sync}}, a : \mathtt{I}(G_{\mathsf{sync}}[S]); \Delta'_{\mathsf{sync}}, \{s_i[S] : T_{\mathsf{sync}}\}_{i \in I}; D'_{\mathsf{sync}}, \mathbf{c} \rangle$$

where I is the set of indexes for the sessions using G_{sync}. This specification is *not* asynchronous because Θ_{sync} can accept trace

$$s_1[C,S]?\mathrm{req}(\varepsilon) \cdot s_1[S,C]!\mathrm{ans}\langle 1\rangle \cdot s_2[C,S]?\mathrm{req}(\varepsilon) s_2[S,C]!\mathrm{ans}\langle 2\rangle$$

but cannot accept the trace below which is permuted from the trace above

$$s_1[C,S]?\mathrm{req}(\varepsilon) \cdot s_2[C,S]?\mathrm{req}(\varepsilon) \cdot s_2[S,C]!\mathrm{ans}\langle 2\rangle \cdot s_1[S,C]!\mathrm{ans}\langle 1\rangle$$

On the other hand, checking asynchrony by Definition 8.5.7 means we should verify this property for all traces, which are usually infinitely many. Later we shall introduce the decidable methods by which we can approve the asynchrony of, for example, Θ_{ass} and all the corresponding specifications that apply stateful protocols $G_{\mathsf{pcs}}/G_{\mathsf{ivc}}$ and G_{async}.

The following characterisation says that, if a synchronous observer recognises that P conforms to Θ, i.e. if P conforms to Θ synchronously, then an asynchronous observer will also do the same.

Proposition 8.5.9. Θ is asynchronous iff, for each P, $P \models_{\mathtt{sync}} \Theta$ implies $P \models_{\mathtt{async}} \Theta$.

Proof. For ($\Rightarrow$), assume $P \models_{\mathtt{sync}} \Theta$ and Θ is asynchronous. By definition, $P \models_{\mathtt{sync}} \Theta$ means $\mathsf{Obs}_s(P) \subset \mathtt{trace}(\Theta)$. Since Θ is asynchronous, by Definition 8.5.7, all legal permutation variants of $\mathtt{trace}(\Theta)$ are in $\mathtt{trace}(\Theta)$. This implies all legal permutation variants of $\mathsf{Obs}_s(P)$, which is $\mathsf{Obs}_a(P)$, are in $\mathtt{trace}(\Theta)$. Thus $P \models_{\mathtt{async}} \Theta$ as required.

For ($\Leftarrow$), since $\forall P, P \models_{\mathtt{sync}} \Theta$ implies $P \models_{\mathtt{async}} \Theta$, we know

$$\forall P, \forall \mathtt{s} \in \mathsf{Obs}_s(P) \subset \mathtt{trace}(\Theta) \text{ and } \mathtt{s} \curvearrowright \mathtt{s}' \text{ implies } \mathtt{s}' \in \mathsf{Obs}_a(P) \subset \mathtt{trace}(\Theta).$$

By Definition 8.5.7, Θ is asynchronous. ■

Let $\mathtt{I}(\mathtt{s})$ be the sequence of input actions of $\mathtt{s}$, $\mathtt{O}(\mathtt{s})$ be the sequence of output actions of $\mathtt{s}$, and $\mathtt{V}(\mathtt{s})$ be the sequence of values carried in $\mathtt{s}$. Let $\mathtt{num}(\mathtt{s})$ be the length of $\mathtt{s}$, and $\mathtt{prefix}(\mathtt{s})$ be the set of prefixes of $\mathtt{s}$. Let $\mathtt{ses}(\mathtt{s})$ be the set of sessions occuring in trace $\mathtt{s}$, and, similarly, $\mathtt{ses}(\Theta)$ be the set of sessions involving in Θ, i.e. those sessions obeying

Θ. Note that, the size of set $\mathtt{ses}(\mathtt{s})$ is increasing when new sessions are created by a local process; similarly, the size of set $\mathtt{ses}(\Theta)$ is increasing when new sessions obeying Θ are created. Moreover, let $\mathtt{v}(\ell)$ be the value carried by action ℓ.

When a state is declared in a local specification, since a specification can dynamically verify sequence of actions during runtime, the state may be changed with the incoming or outgoing actions according to the update rules defined in the specification. Therefore, the current value of a state in a specification should be defined with actions of a process because these actions may make the value of state change.

Definition 8.5.10. Define $\mathtt{field}(\Theta)$ as the set of states declared in Θ. If $\mathbf{c} \in \mathtt{field}(\Theta)$, then $\mathbf{c}$ is a state declared in Θ.

Definition 8.5.11 (current value of a state with respect to Θ and actions). Let $\mathbf{f} \in \mathtt{field}(\Theta)$ and $\mathtt{val}(\mathbf{f}, \Theta')$ be the value of state $\mathbf{f}$ when the configuration of specification is Θ'. Define the current value of state $\mathbf{f}$ as

$$\mathtt{current}(\mathbf{f}, \Theta, \mathtt{s}) = \mathtt{val}(\mathbf{f}, \Theta') \text{ where } \Theta \xrightarrow{\mathtt{s}} \Theta'.$$

Based on this definition, $\mathtt{current}(\mathbf{f}, \Theta, \mathtt{s})$ is the current value of state $\mathbf{f}$ according to Θ, which specifies the initial value of $\mathbf{f}$, when actions in $\mathtt{s}$ have happened.

Definition 8.5.12 (initial value of a state w.r.t. Θ). Define $\mathtt{current}(\mathbf{f}, \Theta, \emptyset)$ be the initial value of state $\mathbf{f}$ which is declared in Θ.

The initial value of a state should have been declared in Θ. For example, as $\mathbf{f} \mapsto \mathsf{ini}$ is declared in Θ, we have $\mathtt{current}(\mathbf{f}, \Theta, \emptyset) = \mathsf{ini}$.

Lemma 8.5.13. Assume $\mathbf{c} \in \mathtt{field}(\Theta_{\mathsf{sync}})$ and $\mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \emptyset) = \mathsf{ini}$. Assume every session s_i guided by G_{sync} has been established. Θ_{sync} is defined as below:

$$\begin{aligned}\Theta_{\mathsf{sync}} \;=\; & \langle \mathtt{ser} : \mathtt{I}(G_{\mathsf{sync}}[S]) \;; \\ & \{s_i[S] : C?\mathrm{req}(\varepsilon)\langle\!\langle \mathsf{true} \;; \varepsilon \rangle\!\rangle . C!\mathrm{ans}(x : \mathsf{int})\langle\!\langle x = \mathbf{c} \;;\; \mathbf{c} := \mathbf{c} + 1 \rangle\!\rangle\}_{i \in I} \;; \\ & \mathbf{c} \mapsto \mathsf{ini}\rangle\end{aligned}$$

where I is the set of indexes of sessions. If $\mathtt{s}' \in \mathtt{trace}(\Theta_{\mathsf{sync}})$, then for any $\mathtt{s}$ resulting from permutations $\mathtt{s}' \curvearrowright \mathtt{s}$, $\mathtt{s}$ satisfies the followings:

cond 1. If session $s \in \mathtt{ses}(\mathtt{s})$, then session $s \in \mathtt{ses}(\Theta_{\mathsf{sync}})$.

cond 2. $\forall \mathtt{s}'' \in \mathtt{prefix}(\mathtt{s}),\ \mathsf{num}(\mathtt{I}(\mathtt{s}'')) \geq \mathsf{num}(\mathtt{O}(\mathtt{s}''))$.

cond 3. If $\mathtt{s}_0 \cdot \ell \in \mathtt{s}\ \cap\ \ell = s[S,B]!\mathrm{ans}\langle v\rangle$, then $\exists\ \ell' \in \mathtt{s}_0,\ \ell' = s[B,S]?\mathrm{req}(\varepsilon)$.

cond 4. $\forall \ell, \ell' \in \mathtt{O}(\mathtt{s}),\ \ell \neq \ell'$, implies $\mathtt{v}(\ell) \neq \mathtt{v}(\ell')$.

cond 5. $\forall \ell \in \mathtt{O}(\mathtt{s}''), \mathtt{s}'' \in \mathtt{prefix}(\mathtt{s}), \mathsf{ini} \leq \mathtt{v}(\ell) < \mathsf{num}(\mathtt{I}(\mathtt{s}'')) + \mathsf{ini}$.

Proof. It is proved in Appendix C.1.

Proposition 8.5.14. Assume $\mathbf{c}', \mathbf{log} \in \mathtt{field}(\Theta_{\mathsf{ass}})$ and $\mathtt{current}(\mathbf{c}', \Theta_{\mathsf{ass}}, \emptyset) = \mathsf{ini}' = \mathsf{ini} - 1$, $\mathtt{current}(\mathbf{log}, \Theta_{\mathsf{ass}}, \emptyset) = \{\ \}$. Assume every session s_i guided by G_{assign} has been established. Θ_{ass} is defined below:

$$
\begin{array}{rcl}
\Theta_{\mathsf{ass}} & = & \langle \mathtt{ser} : \mathtt{I}(G_{\mathsf{assign}}[S])\ ; \\
& & \{s_i[S] : C?\mathrm{req}(\varepsilon)\langle\!\langle \mathsf{true}\ ;\ \mathbf{c}' := \mathbf{c}' + 1, \mathbf{log} := \mathbf{log} \cup \{\mathbf{c}'\}\rangle\!\rangle. \\
& & \qquad C!\mathrm{ans}(x : \mathsf{int})\langle\!\langle x \in \mathbf{log}\ ;\ \mathbf{log} := \mathbf{log} \setminus \{x\}\rangle\!\rangle\}_{i \in I}\ ; \\
& & \mathbf{c}' \mapsto \mathsf{ini}', \mathbf{log} \mapsto \{\ \}\rangle
\end{array}
$$

where I is the set of indexes of sessions. $\mathtt{s}$ satisfies all conditions listed in Lemma 8.5.13, *if and only if* $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{ass}})$.

Proof. It is proved in Appendix C.1.

Definition 8.5.15 (the strongest asynchronous specification). Let Θ_1 be the strongest asynchronous specificaiton w.r.t Θ_2 if

1. Θ_1 is asynchronous, and
2. $\forall P.(\mathtt{s} \in \mathsf{Obs}_s(P)\ \cap\ P \models_{\mathsf{sync}} \Theta_{\mathsf{ass}})$ if and only if $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{ass}})$.

Proposition 8.5.16. Θ_{ass} is the strongest specification w.r.t Θ_{sync}.

Proof. Based on Definition 8.5.15, it is proved directly from Lemma 8.5.13 and Proposition 8.5.14.■

Lemma 8.5.17. If $P \models_{\mathsf{sync}} \Theta_{\mathsf{sync}}$, then $\forall \mathtt{s} \in \mathsf{Obs}_a(P)$ implies $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{ass}})$

Proof. Firslty, $P \models_{\mathsf{sync}} \Theta_{\mathsf{sync}}$ means that, for any $\mathtt{s}' \in \mathsf{Obs}_s(P)$, we have $\mathtt{s}' \in \mathtt{trace}(\Theta_{\mathsf{sync}})$, and for any $\mathtt{s} \in \mathsf{Obs}_a(P)$, $\mathtt{s}$ is the result of legal permutations from some $\mathtt{s}' \in \mathsf{Obs}_s(P)$. In other words, for any $\mathtt{s} \in \mathsf{Obs}_a(P)$, there exists $\mathtt{s}'$ such that $\mathtt{s}' \in \mathsf{Obs}_s(P)$ and $\mathtt{s}' \in \mathtt{trace}(\Theta_{\mathsf{sync}})$, then $\mathtt{s}$ can be reached from $\mathtt{s}' \curvearrowright \mathtt{s}$. Based on Lemma 8.5.13, $\mathtt{s}$ satisfies conditions listed there. Since for any $\mathtt{s}$ satisfies these conditions, by Lemma 8.5.14, it implies $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{ass}})$, thus $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{ass}})$. ■

Definition 8.5.18 (input-output alternating sequence). Let ℓ_i^{in} be the ith input action and ℓ_i^{out} be the ith output action. $\mathtt{s}$ is called an input-output alternating sequence if it is of the shape

$$\mathtt{s} = \ell_1^{in} \cdot \ell_1^{out} \cdot \ell_2^{in} \cdot \ell_2^{out} .....$$

where ℓ_1^{in} is the first action and the sequence of actions will end up with ℓ_k^{in} or $\ell_k^{in} \cdot \ell_k^{out}$ for some $k \geq 1$.

Proposition 8.5.19. If $P \models_{\mathsf{sync}} \Theta_{\mathsf{ass}}$, then $P \models_{\mathsf{async}} \Theta_{\mathsf{ass}}$.

Proof. It can be proved by proving that Θ_{ass} is asynchronous through proving it is commutative with the definitions and propositions introduced later. Another direct proof without proving that Θ_{ass} is asynchronous is provided in Appendix C.1.

Proposition 8.5.20. Assume $\mathtt{s}$ is an input-output alternating sequence. If $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{ass}})$, then $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{sync}})$.

Proof. Please see Appendix C.1.

The additional lemmas for proving Propositions 8.5.19 and 8.5.20 are in Appendix C. Although Proposition 8.5.20 says that G_{assign} is an asynchronous verifiable specification that can realise synchronous behaviours w.r.t Θ_{sync}, the other way round is not true. The following example illustrates this point.

Example 8.5.21. Assume the initial value of state $\mathbf{c} \in \Theta_{\mathsf{ass}}$ is empty set (i.e. $\mathbf{c} \mapsto \{\}$) and the initial value of state $\mathbf{t}$ is 1. Also, assume the initial value of $\mathbf{c} \in \Theta_{\mathsf{sync}}$ is 1.

Consider a trace $\mathtt{s}$:

$$s_1[C,S]?\mathrm{req}(\varepsilon) \cdot s_2[C,S]?\mathrm{req}(\varepsilon) \cdot s_1[S,C]!\mathrm{ans}\langle 2 \rangle$$

then $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{ass}})$ but $\mathtt{s} \notin \mathsf{Obs}_a(P)$ where $P \models_{\mathsf{sync}} \Theta_{\mathsf{sync}}$ because there does not exist $\mathtt{s}' \in \mathtt{trace}(\Theta_{\mathsf{sync}})$ such that $\mathtt{s}' \curvearrowright \mathtt{s}$.

The next result says that asynchronous verifiability is consistent with the asynchronous trace equivalence. Below let $P \approx_{\mathsf{async}} Q$ mean $\mathsf{Obs}_a(P) = \mathsf{Obs}_a(Q)$. In (88), we have shown how $\approx_{\mathsf{async}}$ (but not its synchronous counterpart) can be used for non-trivial optimising transformation.

Proposition 8.5.22. If $P \approx_{\mathsf{async}} Q$ and $P \models_{\mathsf{async}} \Theta$ then $Q \models_{\mathsf{async}} \Theta$.

Proof. Assume $P \approx_{\mathtt{async}} Q$ and $P \models_{\mathtt{async}} \Theta$. We first show $\mathsf{Obs}_a(Q) \subset \mathtt{trace}(\Theta)$. By assumption we have:

1. If $\mathtt{s} \in \mathsf{Obs}_a(Q)$, then $\mathtt{s} \in \mathsf{Obs}_a(P)$ based on $P \approx_{\mathtt{async}} Q$.
2. If $\mathtt{s} \in \mathsf{Obs}_a(P)$, then $\mathtt{s} \in \mathtt{trace}(\Theta)$ based on $P \models_{\mathtt{async}} \Theta$.

Hence if $\mathtt{s} \in \mathsf{Obs}_a(Q)$, then $\mathtt{s} \in \mathtt{trace}(\Theta)$, as required.

We next show input consistency. Suppose $\mathtt{s} \in \mathsf{Obs}_a(Q)$ and $\mathtt{s} \cdot \ell \in \mathtt{trace}(\Theta)$ where ℓ is input. $P \approx_{\mathtt{async}} Q$ implies $\mathtt{s} \in \mathsf{Obs}_a(P)$, and $P \models_{\mathtt{async}} \Theta$ implies $\mathtt{s} \cdot \ell \in \mathsf{Obs}_a(P)$. Again by $P \approx_{\mathtt{async}} Q$, we obtain $\mathtt{s} \cdot \ell \in \mathsf{Obs}_a(Q)$ as required. This concludes the proof.■

8.5.3 Commutativity

A basic issue in Definition 8.5.7 and its characterisation in Proposition 8.5.9 is that they do not directly mention the (intensional) structure of specifications. Thus it does not offer engineers insights as to how one may design her/his specifications. Extending the usage of the term in (81), we may call a criteria for specifications which a designer can use for ensuring robustness w.r.t. asynchrony, *healthiness condition*. The following definition is a first step towards such a criteria.

Definition 8.5.23 (confluence). Θ is *confluent* if, whenever $\Theta \xrightarrow{\mathtt{s}} \Theta'$, if $\Theta' \xrightarrow{\ell_1\ell_2} \Theta''$ and $\ell_2\ell_1 \frown \ell_1\ell_2$, then $\Theta' \xrightarrow{\ell_2\ell_1} \Theta''$ again.

I.e. the specification accepts the same sequence of values regardless of legal permutations and the resulting states are the same. Immediately confluence means asynchrony.

Lemma 8.5.24. Θ is asynchronous iff $\mathtt{s} \cdot \ell_1 \cdot \ell_2 \in \mathtt{trace}(\Theta)$ and $\ell_1 \cdot \ell_2 \frown \ell_2 \cdot \ell_1$ imply $\mathtt{s} \cdot \ell_2 \cdot \ell_1 \in \mathtt{trace}(\Theta)$ for each $\mathtt{s}$, ℓ_1 and ℓ_2.

Proof. For ($\Rightarrow$), it is trivial by Definitions 8.5.7 and 8.5.23. For ($\Leftarrow$), we want to prove that all cases should satisfy Definition 8.5.7, thus Θ is asynchronous. The reasonings are as follows:

1. If $\mathtt{s} = \emptyset$, which is not permutable, then $\forall \ell_1\ell_2 \in \mathtt{trace}(\Theta), \ell_1\ell_2 \frown \ell_2\ell_1 \in \mathtt{trace}(\Theta)$ satisfies Definition 8.5.7.

2. If $\mathtt{s} \neq \emptyset$, several cases are analysed below:

   (1) If $\mathtt{s} \not\frown$, which means $\mathtt{s}$ is not permutable with any other sequence of actions,

(a) when neither ℓ_1 nor ℓ_2 can permute with $\mathtt{s}$, then the only permutable actions in $\mathtt{s}\cdot\ell_1\cdot\ell_2 \in \mathtt{trace}(\Theta)$ is $\ell_1\cdot\ell_2 \frown \ell_2\cdot\ell_1$, and $\mathtt{s}\cdot\ell_2\cdot\ell_1 \in \mathtt{trace}(\Theta)$. It satisfies Definition 8.5.7.

(b) when there exists $\ell'_1,..,\ell'_j \in \mathtt{s}$ such that ℓ_2 (or ℓ_1) is permutable with $\ell'_1,..,\ell'_j$. Note that, since permutations are done one by one through legal unit permutation, $\mathtt{s}$ should be of the following shape:

$$\mathtt{s} = \mathtt{s}_0\cdot\ell'_1..\ell'_{j-1}\cdot\ell'_j,$$

where $\mathtt{s}_0$ is not permutable at all (i.e. it means that $\mathtt{s}_0$ is empty, or actions in $\mathtt{s}_0$ are not permutable with each other and no action in $\mathtt{s}_0$ is permutable with any action of $\ell'_1,..,\ell'_j$.) Assume that is ℓ_2 can permute with $\ell'_1,..,\ell'_{j-1}$. Then

$$\begin{array}{l}\mathtt{s}_0\cdot\ell'_1..\ell'_{j-1}\cdot\ell'_j\cdot\ell_1\cdot\ell_2 \frown\\ \mathtt{s}_0\cdot\ell'_1..\ell'_{j-1}\cdot\ell'_j\cdot\ell_2\cdot\ell_1 \frown\\ \mathtt{s}_0\cdot\ell'_1..\ell'_{j-1}\cdot\ell_2\cdot\ell'_j\cdot\ell_1\end{array}$$

Let $\mathtt{s}_0\cdot\ell'_1..\ell'_{j-1}\cdot\ell_2 = \mathtt{s}'\cdot\ell_2$. It is either

(i) ℓ_2 can permute with some actions in $\ell'_1..\ell'_{j-1}$ then it is still under the analysis of case 2.(1)(b), or

(ii) ℓ_2 has permuted to all permutable actions in $\mathtt{s}'$ until $\mathtt{s}_0$, which is not permutable at all. Then it is case 2.(1)(a) that satisfies Definition 8.5.7.

Similarly, if $\ell_2\cdot\ell_1 \frown \ell_1\cdot\ell_2$ and ℓ_1 can permute with some actions in $\mathtt{s}$, it is again under the analysis of case 2.(1)(b).

(2) If there exists $\mathtt{s}'$ such that $\mathtt{s} \frown \mathtt{s}'$, $\mathtt{s}$ should be of the following shape:

$$\mathtt{s} = \mathtt{s}'_1\cdot\ell''_1\cdot\ell''_2\cdot\mathtt{s}'_2,$$

where $\ell''_1\cdot\ell''_2 \frown \ell''_2\cdot\ell''_1$ and actions in $\mathtt{s}'_2$ are not permutable with each other and no action in $\mathtt{s}'_2$ is permutable with any action in $\mathtt{s}'_1\cdot\ell''_1\cdot\ell''_2$. Then consider

$$\mathtt{s}'_1\cdot\ell''_1\cdot\ell''_2\cdot\mathtt{s}'_2\ell_1\cdot\ell_2 \frown \mathtt{s}'_1\cdot\ell''_1\cdot\ell''_2\cdot\mathtt{s}'_2\ell_2\cdot\ell_1$$

It is either

(a) ℓ_2 is not permutable with $\mathtt{s}'_2$, then we only consider the part $\mathtt{s}'_1 \cdot \ell''_1 \cdot \ell''_2$, since $\mathtt{s} \cdot \ell_1 \cdot \ell_2 \in \mathtt{trace}(\Theta)$ implies $\mathtt{s} \in \mathtt{trace}(\Theta)$, which again implies $\mathtt{s}'_1 \cdot \ell''_1 \cdot \ell''_2 \in \mathtt{trace}(\Theta)$, so that $\mathtt{s}'_1 \cdot \ell''_2 \cdot \ell''_1 \in \mathtt{trace}(\Theta)$, and this case (i.e. $\mathtt{s}'_1 \cdot \ell''_1 \cdot \ell''_2$) is under the analysis of case 2.(1)(b).

(b) ℓ_2 is permutable with $\mathtt{s}'_1 \cdot \ell''_1 \cdot \ell''_2 \cdot \mathtt{s}'_2$. Then it is again in case 2.(1)(b).

Since all situations in 2.(1)(b) will finally go to 2.(1)(b)(ii), which means when the permutation terminates, it satisfies Definition 8.5.7. Overall, all possible cases satisfy Definition 8.5.7, thus Θ is asynchronous. ■

Proposition 8.5.25. If Θ is confluent, then it is asynchronous.

Proof. Directly from Definition 8.5.23 and Lemma 8.5.24. ■

Note that the other way round is not true. The example below illustrates this point.

Example 8.5.26. Assume the initial values of states $\mathbf{c}$ and $\mathbf{t}$ are both 0, and the local SP for role p in session s is:

$$s[p] : q_1!\mathrm{ans}(\varepsilon)\langle\!\langle \mathsf{true}; \mathbf{c} = 50 \rangle\!\rangle \ . \ q_2!\mathrm{ans}(\varepsilon)\langle\!\langle \mathsf{true}; \mathbf{t} = \mathbf{c} + 10 \rangle\!\rangle$$

Let $\ell_1 = s[p, q_1]!\mathrm{ans}\langle\varepsilon\rangle$ and $\ell_2 = s[p, q_2]!\mathrm{ans}\langle\varepsilon\rangle$. Then $\ell_1 \cdot \ell_2$ makes state $\mathbf{t}$ be 60, but $\ell_2 \cdot \ell_1$ makes state $\mathbf{t}$ be 10.

One can easily find a specification which is not confluent (for example, if a specification just does the same counting as the one defined with stateful protocol G_{sync}). To check confluence, we still need to consider all possible transition derivatives of Θ. However we can observe that, in such a derivative, *the obligations used to check confluence are already present in* Θ. This suggests us to only look at the obligations occurring in Θ and check their commutativity w.r.t. their legal unit permutations. This method demands designers to look only at Θ, so that it clearly helps her/his design process. The method treats a predicate and an update in an obligation as functions (operations) on state, as follows. Let $\dagger \in \{?, !\}$.

Definition 8.5.27 (predicate/update functions). Let $\xi \stackrel{\text{def}}{=} r \dagger l(x : S)\langle\!\langle A; E \rangle\!\rangle$ with the associated state D whose domain is $\mathbf{f}_1, ..., \mathbf{f}_n$. W.l.o.g. we regard E to be a

simultaneous substitution of the form $\mathbf{f}_1 := e_1, ..., \mathbf{f}_n := e_n$. Then we define:

$$\texttt{pred}(\xi) \stackrel{\text{def}}{=} \lambda x, \mathbf{f}_1, ..., \mathbf{f}_n.A \qquad \texttt{upd}(\xi) \stackrel{\text{def}}{=} \lambda x, \mathbf{f}_1, ..., \mathbf{f}_n.\langle e_1, .., e_n\rangle$$

We call $\texttt{pred}(\xi)$ (resp. $\texttt{upd}(\xi)$) the *predicate function* (resp. *update function*) of ξ.

Example 8.5.28. Below we project G_{sync} and G_{assign} (all from Section 8.1) to the server. For simplicity we assume its local state only consists of those fields specified in global SP.

$$G_{\mathsf{sync}} \upharpoonright S \;=\; B?\text{req}(\varepsilon)\langle\!\langle \mathsf{true}; \varepsilon\rangle\!\rangle \;.\; B!\text{ans}(x : \mathsf{int})\langle\!\langle x = \mathbf{c} \;;\; \mathbf{c} := \mathbf{c} + 1\rangle\!\rangle$$

$$\begin{aligned} G_{\mathsf{assign}} \upharpoonright S \;=\; & B?\text{req}(\varepsilon)\langle\!\langle \mathsf{true}; \mathbf{t} := \mathbf{t} + 1 \;\; \mathbf{c} := \mathbf{c} \cup \{\mathbf{t}\}\rangle\!\rangle \;. \\ & B!\text{ans}(x : \mathsf{int})\langle\!\langle x \in \mathbf{c} \;;\; \mathbf{c} := \mathbf{c} \setminus \{x\}\rangle\!\rangle \end{aligned}$$

Then the following table gives the functions induced by obligations in these local types.

	input	output
$G_{\mathsf{sync}} \upharpoonright S$	$\xi_0 \stackrel{\text{def}}{=} B?\text{req}(\varepsilon)\langle\!\langle \mathsf{true}; \varepsilon\rangle\!\rangle$	$\xi_1 \stackrel{\text{def}}{=} B!\text{ans}(x : \mathsf{int})\langle\!\langle x = \mathbf{c}; \mathbf{c} := \mathbf{c} + 1\rangle\!\rangle$
	$\texttt{pred}(\xi_0) \stackrel{\text{def}}{=} \lambda\varepsilon, \mathbf{c}.(\mathsf{true})$	$\texttt{pred}(\xi_1) \stackrel{\text{def}}{=} \lambda x, \mathbf{c}.(x = \mathbf{c})$
	$\texttt{upd}(\xi_0) \stackrel{\text{def}}{=} \lambda\varepsilon, \mathbf{c}.\langle\!\langle\varepsilon\rangle\!\rangle$	$\texttt{upd}(\xi_1) \stackrel{\text{def}}{=} \lambda x, \mathbf{c}.\langle\!\langle \mathbf{c} + 1\rangle\!\rangle$
$G_{\mathsf{assign}} \upharpoonright S$	$\xi_2 \stackrel{\text{def}}{=} B?\text{req}(\varepsilon)\langle\!\langle \mathsf{true}; \mathbf{t} := \mathbf{t} + 1 \;\; \mathbf{c} := \mathbf{c} \cup \{\mathbf{t}\}\rangle\!\rangle$	$\xi_3 \stackrel{\text{def}}{=} B!\text{ans}(x : \mathsf{int})\langle\!\langle x \in \mathbf{c} \;;\; \mathbf{c} := \mathbf{c} \setminus \{x\}\rangle\!\rangle$
	$\texttt{pred}(\xi_2) \stackrel{\text{def}}{=} \lambda\varepsilon, \mathbf{c}.(\mathsf{true})$	$\texttt{pred}(\xi_3) \stackrel{\text{def}}{=} \lambda x, \mathbf{c}.(x \in \mathbf{c})$
	$\texttt{upd}(\xi_2) \stackrel{\text{def}}{=} \lambda\varepsilon, \mathbf{c}.\langle\!\langle \mathbf{t} + 1 \;\; \mathbf{c} \cup \{\mathbf{t}\}\rangle\!\rangle$	$\texttt{upd}(\xi_3) \stackrel{\text{def}}{=} \lambda x, \mathbf{c}.\langle\!\langle \mathbf{c} \setminus \{x\}\rangle\!\rangle$

Once we can treat obligations as operations on state, we can define their commutativity. Since the commutativity we need is asymmetric (corresponding to asymmetric permutations induced by asynchrony, cf. Definition 8.4.12), we define semi-commutativity, which plays a key role in validating specifications later. A precursor of the following construction in a different setting is found in (52) (see Chapter 9 for discussions).

Definition 8.5.29 (semi-commutativity). Assume w.l.o.g., ξ_i and ξ_j use $\mathbf{f}$ as the field. Then we say ξ_i *commutes over* ξ_j if, for any message values v_i and v_j (for ξ_i and ξ_j), and any initial state w (for $\mathbf{f}$), the following conditions hold: If $\texttt{pred}(\xi_i)(v_i, w)$ and $\texttt{pred}(\xi_j)(v_j, \texttt{upd}(\xi_i)(v_i, w))$ are both true, then

1. $\texttt{pred}(\xi_j)(v_j, w)$ and $\texttt{pred}(\xi_i)(v_i, \texttt{upd}(\xi_j)(v_j, w))$ are both true.

2. $\mathtt{upd}(\xi_j)(v_j, \mathtt{upd}(\xi_i)(v_i, w)) = \mathtt{upd}(\xi_i)(v_i, \mathtt{upd}(\xi_j)(v_j, w))$.

If ξ_i commutes over ξ_j and vice versa, then we say ξ_i *and* ξ_j *are commutative.*

Note that the *if* statement says that it should satisfy the conditions above for *any* kind of initial values. I will explain the reason with Example 8.5.36 after Definition 8.5.31 and Proposition 8.5.34 are introduced.

Example 8.5.30. We show ξ_1 in Example 8.5.28 does not commute over itself. Let $\mathbf{f} = \mathbf{c}$. Then $\mathtt{pred}(\xi_1)(1,1)$, $\mathtt{pred}(\xi_1)(2, \mathtt{upd}(\xi_1)(1,1))$ and $\mathtt{pred}(\xi_1)(2,2)$ are all true, however $\mathtt{pred}(\xi_1)(2,1) = \mathsf{false}$. Similarly, ξ_0 does not commute over ξ_1 (however ξ_0, ξ_0 are commutative).

Using this notion, the healthiness condition for asynchronous specification can be concisely stated as follows. Below we say an obligation is *usable in* Θ if it occurs in a local SPin Θ or in the projection of a global SPin Θ to its potentially local role, where by "potentially local" we mean that the role has a potential to be played locally (e.g. for the global SPcarried by an input shared channel type (i.e. $a(s[p] : G)$), only the specified role is potentially local).

Definition 8.5.31 (commutativity). Given Θ, let $\xi_1, .., \xi_n$ be all the obligations usable in Θ. Then we say Θ is *commutative* if the following conditions hold:

1. For (possibly identical) ξ_1' and ξ_2' from $\{\xi_1, .., \xi_n\}$, if both are inputs or both are outputs, then ξ_1' and ξ_2' are commutative.
2. For distinct ξ_1' and ξ_2' from $\{\xi_1, .., \xi_n\}$, if ξ_1' is an output and ξ_2' is an input then ξ_1' commutes over ξ_2'.

Possibly identical obligations means that it can be two obligations which are exactly the same, e.g. analysing ξ, ξ. In particular when there is only one obligation in Θ, this obligation, say ξ, should be analysed by itself ξ, ξ to know whether it commutes over itself.

Note that, based on Definition 8.5.29, Definition 8.5.31 states that, Θ is commutative if, for *any kind of initial values* in Θ, it satisfies the conditions given above. It is very important that it is for *any* initial value. Even though, assume a specification Θ is given with initial value w, Θ satisfies all conditions above with w but not with another initial value, say $w' \neq w$, then Θ is not commutative. I.e. Θ is action confluent when

all obligations used in the specifications for the target process commute over each other up to legal permutations.

Before proving "Θ is commutative implies Θ is confluent", the following lemma is introduced.

Lemma 8.5.32. Assume $\Theta \xrightarrow{\mathtt{s}} \Theta'$ for some $\mathtt{s}$. If Θ is commutative, then Θ' is commutative.

Proof. For ($\Rightarrow$), because $\Theta' \subset \Theta$:

1. For any obligation $\xi \in \Theta'$, $\xi \in \Theta'$ implies $\xi \in \Theta$, where Θ is commutative so that ξ itself is commutative.

2. For any two obligations $\xi, \xi' \in \Theta$ but $\xi' \notin \Theta'$ (an obligation may be finished since the conversation is finished. See Figure 8.3, the LTS of specifications), since ξ itself should be commutative due to $\xi \in \Theta$ and Θ is commutative, ξ is still commutative in Θ' even it is alone.

Thus Θ' is commutative with new values of states resulting from $\Theta \xrightarrow{\mathtt{s}} \Theta'$ if Θ is commutative. ■

Note that the other way round (i.e. $\Leftarrow$) is not true because, while Θ' is derived from Θ, Θ can have more obligations than what Θ' does. Θ' is commutative cannot imply that Θ is commutative.

Remark 8.5.33. Definitions 8.5.27 and 8.5.29 imply every obligation ξ corresponds to a valid action, and vice versa.

We have soundness characterisation for asynchronously verifiable specifications:

Theorem 8.5.34 (soundness characterisation: commutativity). If Θ is commutative then it is confluent (hence asynchronous).

Proof. Assume that Θ is commutative is given, let $\Theta \xrightarrow{\mathtt{s}} \Theta'$ for some $\mathtt{s}$. Assume $\Theta' \xrightarrow{\ell_1\ell_2} \Theta''$ for some ℓ_1 and ℓ_2, and $\ell_1{\cdot}\ell_2 \frown \ell_2{\cdot}\ell_1$. We will show that $\Theta' \xrightarrow{\ell_2\ell_1} \Theta'''$ and $\Theta'' = \Theta'''$. Assume the value carried by ℓ_i is v_i, $i = \{1, 2\}$. Since Θ' is commutative based on Lemma 8.5.32:

1. When both of ℓ_1 and ℓ_2 are for output (resp. for input), these actions correspond to (possibly identical) obligations ξ_1 and ξ_2 from Θ', which are both for outputs

(resp. for inputs). Assume the vector of values of states $\tilde{\mathbf{f}} \in \xi_1 \cup \xi_2$ is $\tilde{w}$, then because Θ' is commutative,

$$\mathtt{pred}(\xi_1)(v_1, \tilde{w}) = \mathtt{pred}(\xi_2)(v_2, \mathtt{upd}(\xi_1)(v_1, \tilde{w})) = \mathsf{true}$$

implies

$$\mathtt{pred}(\xi_2)(v_2, \tilde{w}) = \mathtt{pred}(\xi_1)(v_1, \mathtt{upd}(\xi_2)(v_2, \tilde{w})) = \mathsf{true}$$

which means $\Theta' \xrightarrow{\ell_2\ell_1} \Theta'''$ for some Θ''', and moreover,

$$\mathtt{upd}(\xi_1)(v_1, \mathtt{upd}(\xi_2)(v_2, \tilde{w})) = \mathtt{upd}(\xi_2)(v_2, \mathtt{upd}(\xi_1)(v_1, \tilde{w})) = \tilde{w'}$$

which means $\ell_1 \cdot \ell_2$ and $\ell_2 \cdot \ell_1$ consume the same obligations and they reach the same value of states $\tilde{w'}$, so that we have $\Theta'' = \Theta'''$.

2. When ℓ_1 is for output and ℓ_2 is for input, these actions correspond to distinct obligations ξ_1 and ξ_2 from Θ', which are separately for output and for input. Assume the vector of values of states $\tilde{\mathbf{f}} \in \xi_1 \cup \xi_2$ is $\tilde{w}$, then

$$\mathtt{pred}(\xi_1)(v_1, \tilde{w}) = \mathtt{pred}(\xi_2)(v_2, \mathtt{upd}(\xi_1)(v_1, \tilde{w})) = \mathsf{true}$$

implies

$$\mathtt{pred}(\xi_2)(v_2, \tilde{w}) = \mathtt{pred}(\xi_1)(v_1, \mathtt{upd}(\xi_2)(v_2, \tilde{w})) = \mathsf{true}$$

which means $\Theta' \xrightarrow{\ell_2\ell_1} \Theta'''$ for some Θ''', and moreover,

$$\mathtt{upd}(\xi_1)(v_1, \mathtt{upd}(\xi_2)(v_2, \tilde{w})) = \mathtt{upd}(\xi_2)(v_2, \mathtt{upd}(\xi_1)(v_1, \tilde{w})) = \tilde{w'}.$$

which means $\ell_1 \cdot \ell_2$ and $\ell_2 \cdot \ell_1$ consume the same obligations and they reach the same value of states $\tilde{w'}$, so that we have $\Theta'' = \Theta'''$.■

Note that the other way round is not true. The completeness characterisation is however more difficult because we need to know the invariants of initial value of the state which is asserted by logical formula and stipulated by update rules. The following examples explain the reasons.

Example 8.5.35. Assume Θ has the following local SP, called T. The initial value of $\mathbf{c} \in \Theta$ is 2, and $\mathbf{c}$ will only be updated by the actions defined in T. It shows that Θ is

confluent but not commutative.

$$\begin{array}{rcl} T & = & B!\{\mathrm{ans}_1(\varepsilon)\langle\!\langle \mathsf{true}\ ;\ \mathbf{c} := \mathbf{c}+1\rangle\!\rangle\ , \\ & & \quad \mathrm{ans}_2(\varepsilon)\langle\!\langle c < 2\ ;\ \mathbf{c} := \mathbf{c}+\mathbf{c}\rangle\!\rangle\}. \\ & & \mathsf{end} \end{array}$$

It is confluent because based on Definition 8.5.23, we assume Θ starts from initial value of $\mathbf{c}$, which is 2. Whenever $\Theta \xrightarrow{\mathbf{s}} \Theta' \xrightarrow{\ell_1\ell_2} \Theta''$ and $\ell_1{\cdot}\ell_2 \frown \ell_2{\cdot}\ell_1$, $\Theta \xrightarrow{\mathbf{s}} \Theta' \xrightarrow{\ell_2\ell_1} \Theta''$ is true because ℓ_1 or ℓ_2 never be with branch ans_2; otherwise it becomes $\Theta' \not\xrightarrow{\ell_1\ell_2}$, which is not the case considered.

On the other hand, because Definition 8.5.31 asks that Θ is commutative if it satisfies the commutativity conditions for *any kind of initial value* of $\mathbf{c}$, we know that it does not satisfy the conditions immediately when the initial value of $\mathbf{c}$ is smaller than 2 (i.e., 1, 0, -1, -2, ...). Thus Θ is not commutative.

The main motivation to have property of *commutativity* for Θ is that commutativity provides a *soundness characterisation*, and a more easily verifiable way to decide if Θ is confluent, which automatically implies it is asynchronous-verifiable.

Comparing Definition 8.5.23 and Definition 8.5.31, readers may feel curious about why *commutativity* should hold for any possible initial value of a state $\mathbf{f}$. The reason is because, generally, we do not know what the *invariants* of the initial values of the states of a given Θ are. To ensure that a Θ is commutative, we need to know if it satisfies all conditions for all possible initial values. The following example illustrates this point:

Example 8.5.36. Consider a specification, Θ, having a state $\mathbf{c}$ with initial value 8 and the following local SP, called T. Assume T is for endpoint $s[q]$ and $\mathbf{c}$ will only be updated by the actions defined in T.

$$\begin{array}{rcl} T & = & p_1!\mathrm{req}_1(\varepsilon)\langle\!\langle \mathbf{c} < 10; \varepsilon\rangle\!\rangle. \\ & & p_2!\mathrm{req}_2(\varepsilon)\langle\!\langle \mathbf{c} < 10; \mathbf{c} := \mathbf{c}+1\rangle\!\rangle. \\ & & \mathsf{end} \end{array}$$

If commutativity is defined only for a particular initial value of $\mathbf{c} \in \Theta$, which is 8, then Θ is commutative; however, Θ is *not confluent*. Let $\ell_1 = s[q, p_1]!\mathrm{req}_1\langle\varepsilon\rangle$ and $\ell_2 = s[q, p_2]!\mathrm{req}_2\langle\varepsilon\rangle$, when several actions in trace $\mathbf{s}$ update $\mathbf{c}$ and $\mathbf{c}$ becomes 9, and

$$\Theta \xrightarrow{\mathbf{s}} \Theta' \xrightarrow{\ell_1\ell_2} \Theta''$$

is achieved, $\ell_1{\cdot}\ell_2 \frown \ell_2{\cdot}\ell_1$, but $\Theta' \not\xrightarrow{\ell_2\ell_1}$.

This is because the value of $\mathbf{c}$ is changing as actions happen. If Definition 8.5.31 only considers particular initial value of $\mathbf{c}$, then

1. the checking of commutativity is, however, done by a snapshot for any possible pair $\xi_i, \xi_j \in \Theta$. It can not reflect the real situation of the changes of value of $\mathbf{c}$;
2. or, we know exactly what the invariant of Θ is; e.g. the initial value of $\mathbf{c}$ should be bigger than or equal to 10. With this invariant, Θ is confluent. But generally, to find out the invariant(s) of a specification is difficult, which will be part of the future works.

Thus, we generate the soundness characterisation with the following concept:

Remark 8.5.37. Whenever any possible snapshot is taken with any kind of initial value of $\mathbf{c}$, Θ satisfies the conditions defined in Definition 8.5.31 then Θ is commutative.

As mentioned in point 2 above, the method of checking commutativity can be strengthened by adding an invariant (including correlation among states) in state and checking that invariant continues to hold at each step. We can now show all our example specifications with the stateful protocols except the one induced by G_{sync} is asynchronous. Below, let Θ_{async}'s shared environment contains $\mathtt{server} : \mathtt{I}(G_{\mathsf{async}}[S])$, and let its data storage contains $\mathbf{c} \mapsto \{\}$.

$$\begin{array}{rcl} \Theta_{\mathsf{async}} & = & \langle \mathsf{ser} : \mathtt{I}(G_{\mathsf{async}}[S])); \\ & & \{s_j[S] : C?\mathrm{req}(\varepsilon)\langle\!\langle \mathsf{true}\ ; \varepsilon \rangle\!\rangle.C!\mathrm{ans}(x : \mathsf{int})\langle\!\langle x \notin \mathbf{c};\ \mathbf{c} := \mathbf{c} \cup \{x\} \rangle\!\rangle\}_{j \in J}; \\ & & \mathbf{c} \mapsto \{\ \ \} \rangle \end{array}$$

where J is the set of indexes of sessions obeying G_{async}. Then we have the following proposition:

Proposition 8.5.38. Θ_{async} and Θ_{ass} at server are both commutative, hence asynchronous.

Proof. Commutativity of operations in Θ_{async} is immediate. For Θ_{ass}, we use ξ_2 and ξ_3 from Example 8.5.28. Then ξ_3 easily commutes over itself, because if $\mathtt{pred}(\xi_3)(v, \mathbf{c}) = \mathsf{true}$ and $\mathtt{pred}(\xi_3)(v', \mathtt{upd}(\xi_3)(v, \mathbf{c})) = \mathtt{pred}(\xi_3)(v', \mathbf{c} \setminus \{v\}) = \mathsf{true}$, so that

$$\mathtt{pred}(\xi_3)(v', \mathbf{c}) = \mathsf{true}.$$

When $v \neq v'$, we clearly have

$$\mathtt{pred}(\xi_3)(v, \mathtt{upd}(\xi_3)(v', \mathbf{c})) = \mathtt{pred}(\xi_3)(v, \mathbf{c} \setminus \{v'\}) = \mathsf{true}.$$

When $v = v'$, due to $\mathtt{pred}(\xi_3)(v, \mathbf{c}) = \mathsf{true}$ and $\mathtt{pred}(\xi_3)(v', \mathbf{c} \setminus \{v\})$ which together imply that $\mathbf{c}$ contains at least two elements whose values are $v = v'$, we clearly have $\mathtt{pred}(\xi_3)(v, \mathbf{c} \setminus \{v'\}) = \mathsf{true}$. Similarly $\mathtt{upd}(\xi_3)(v', \mathtt{upd}(\xi_3)(v, \mathbf{c})) = \mathtt{upd}(\xi_3)(v', \mathbf{c} \setminus \{v\}) = \mathbf{c} \setminus \{v\} \setminus \{v'\}$ which is equal to $\mathtt{upd}(\xi_3)(v, \mathtt{upd}(\xi_3)(v', \mathbf{c})) = \mathtt{upd}(\xi_3)(v, \mathbf{c} \setminus \{v'\})) = \mathbf{c} \setminus \{v'\} \setminus \{v\}$. We can similarly check ξ_2 is commutative over itself, and ξ_3 commutes over ξ_2 (but not the converse), as required. ■

We can similarly check specifications induced by G_{pcs} and G_{ivc} are commutative.

Compare the definition of commutativity and the permutation rules of Θ, as explained in the beginning of Section 8.4.1, they have different usage:

Remark 8.5.39. Attribute of commutativity is used for developers to know whether a Θ is asynchronous, while the permutation rules (defined in Definition 8.4.4) are used for an observer (e.g. system monitor) to have to observe the incoming and outgoing non-order-preserved actions.

The valuation of commutativity is essentially satisfiability of a formula whose free variables are universally quantified. Thus if the logic (for predicates) we use for our specification language is decidable, commutativity is decidable. In particular:

Proposition 8.5.40. *With the SP language given in Section8.2 restricting operations on integers to be the constant, the addition and the subtraction, then the commutativity of specifications is decidable.*

Proof. By (161), a logic with sets, products and integers with additions is decidable, which subsumes the formulae used in Definition 8.5.29. ■

We discuss practical implications of these results in the conclusion of Section 9.1.3, Chapter 9.

9

Conclusions and Future Topics

9.1 Conclusions and Discussions

This thesis provides fundamental theories for governing real-life large-scale distributed systems from the process calculus to the monitoring mechanism, from session types to stateful specifications. We summarise the main theory chapters, which are Chapter 5, 6, 7, and 8, with conclusions and discussions in the following sections.

9.1.1 The Capability-passing π-calculus

The capability-passing π-calculus proposed in Chapter 5, extends the asynchronous π-calculus by additional primitives of request and accept. These can be used to create sessions linearly and enable the authentication of activities through pattern-matching of the capabilities of proposed participants. The calculus enjoys race-freedom at session roles (86) and reduces the tasks of endpoint access control by filtering unmatched capabilities.

Recently dynamic joining mechanisms have been studied in the conversation-calculus (26) (which uses conversation contexts) and (50) (which is based on roles). These formalisms enable the tracking of participants through roles, but do not enable their controls through capabilities. This thesis formalises asynchronous session creation with capability primitives through passing invitations at shared channels, while (26, 50) formalise joining to the existing session and maintain progress by advanced static type

checking. The result is that this thesis focuses on runtime enforcement of global protocol properties rather than static type checking, and on enforcement of properties by local runtime monitoring. Our framework also incorporates logical assertions, offering more fine-grained logical specifications than the bare protocols representable by types. The adequacy of this formalism is demonstrated by the properties we provide in our discussions of the calculus of dynamic asynchronously monitoring.

9.1.2 The Calculus of Dynamic Asynchronously Monitoring

The session-based specification and monitoring framework in which external monitors evaluate behaviours according to these specifications are respectively studied in Chapters 6 and 7. Session-based specifications, introduced in Chapter 6, provide developers with the ability to build up *their own security protocols*. System developers can program their desired global protocols through an intuitive description language. As an endpoint process starts a session, which generally involves multiple parties, she can select a well-defined protocol from a system library or define a protocol by herself for this session to obey. Then she, as an invitor, invites other parties, called invitees, to join this session playing particular roles defined in the protocol. As an invitee agrees to join the session to play a role, the invitor gives invitees corresponding external monitor guidance of endpoint behaviours, which takes the form of a specification encapsulating a local projection. The endpoint monitor therefore gets the ability to guard invitees' behaviours against the guidance and determines whether their behaviours obey the protocol or not.

Chapter 7 contains a model of distributed monitoring featuring the following elements: (I) endpoint code is possibly ill-behaved; (II) global specifications enable concise global specifications of application-level multi-party protocols; and (III) conformance to global protocols is guaranteed at runtime through local monitoring. This monitoring framework can be seen as a distributed variant of *runtime verification* in the sense of (16, 60, 73). It is light-weight, concerned only with the *observable executions*, and does not aim to give a conclusive analysis about all their possible behaviours. Hence it is applicable to real-world systems, where off-line formal verification and static checking are intractable or impossible due to incomplete information on system components or the dynamic change in requirements.

There are two important principles of the theories of monitoring: *soundness (safety)*, which states that enforcement results in a correct behaviour, and *transparency*, which

requires behaviour of a correct program not be modified (a principle studied for example in an automata framework (43, 143)). Our monitor *suppresses* (108) the illegal actions (as seen in (else) case of $\mathcal{M} \xrightarrow{\ell} \mathcal{M}'$) but neither immediately halts the process nor inserts the correct actions. Incorporating more elaborate reactions, as in edit automata (108, 143) is an interesting future topic. None of the above work can ensure the fidelity to application-level multi-party global protocols for distributed applications with logical properties.

9.1.3 Specifying Asynchronous Stateful Specifications

The characterisation results in Chapter 8 (see Section 8.5) offer not only a decision procedure for a rich subset of specifications, but also a basic insight in the design methodology for asynchronous specifications. In particular they shed light on the use of operations on sets in our examples in Section 8.1. Because checking commutativity relies solely on the obligations occurring in protocols, adding recursion to the syntax. does not change the nature of commutativity checking nor the resulting guarantee.

If Θ is asynchronous and a process behaves properly w.r.t. Θ synchronously, an asynchronous observer will also judge the induced (permuted) trace is proper w.r.t. Θ. It is however easy to see that the converse is *not* true: for example, consider a server that violates Θ_{ass} by responding 2 to the first request, 1 to the second, but these are delayed by asynchrony, leading to a valid trace when they arrive at the remote observer. A key consistency property is that any further legal permutation of this valid trace is again valid. For example, if a system monitor for Server is sitting between Buyer (i.e. as a client) and Server, and if this monitor observes a valid trace of Server against the specification she has, Buyer will observe no worse behaviour. This monotonic attribute gives a basis for an application of the presented framework such as runtime monitoring.

The semantic differences between synchronous and asynchronous communications have been studied for several decades: early works include (7, 45, 84, 90). The permutations associated with asynchronous communication used in Definition 8.4.12 are noted in these works (and implicit in such work as (98)). Their more explicit presentation in the categorical setting is found in (147). There is also a study in component validation based on asynchronous histories such as (130). In spite of these precursors and close technical connection, the existing works (except (21) which however focuses on synchronous specifications and proof rules for their verifications) may not have pointed out

the concrete semantic issues which stateful behavioural specifications and asynchronous observables can induce, and how these issues can be resolved through the interplay between synchronous and asynchronous semantics.

As observed in Section 8.5.3, a close analogue of the commutativity of operations used for our characterisation result (Definition 8.5.29) appears in (52), where the authors study a method for checking commutativity (called *diamond connectivity*) of operations with pre-conditions in object-oriented programs, with a view to preventing the simultaneous issuance of these operations when they are not commutative. They do not (aim to) determine a class of specifications suitable for asynchronously communicating processes.

In contrast, we stipulate a general class of specifications for communicating processes suitable for asynchronous observations, and identify its subclass amenable for automatic verification. We use commutativity to capture asymmetry in asynchronous communications. Moreover, we propose and prove the properties for asserting that an asynchronously verifiable specification is the strongest while one wishes to transform a synchronously verifiable specification to an asynchronously verifiable one .

9.1.4 Concluding Remarks

The achievements of this thesis include (I) establishing formal syntax and semantics for the session-based dynamic monitoring with control capabilities, (II) proposing a fine-grained process calculus for linear asynchronous session creation, (III) exploring the differences between monitoring the behaviours of the local and the global, (IV) enriching theories of session types by embedding endpoints' states into specifications, (V) exploring the differences of observable behaviours between stateful asynchronous and stateful synchronous monitoring, and (VI) proving properties of local and global safety (soundness), transparency, and session fidelity (completeness) for the monitoring, and proposing and proving properties for validating asynchronous specifications. As a cooperative project, the large-scale cyber-infrastructure of OOI (129) is adopting our theories as a blueprint for designing their governance systems. Also, a session-based description language, called Scribble (83, 145), is embedding our specifications into its syntax to realise more controls during programming. For the future, we have important directions as those discussed in our future works (introduced in Section 9.2 later) to explore using this work as a basis. We aim at exploring more theoretical results in

stateful specifications, the adaption of stateful specifications to stateful policy languages, commitments, and the applications of Scribble.

9.2 Future Topics

9.2.1 The Development of Scribble and Monitoring

Scribble (145, 146) is a protocol description language where a protocol is an agreement on how participants interact with each other. Developers can use it to easily code meaningful interaction: the communications among participants become effective, since they know when to expect the other parties to send their data, or whether the other party is ready to receive data it is sending. Scribble makes dynamic distributed monitoring straightforward through projecting the global description, written in Scribble, to every local endpoint as a local specification, and then embedding the generated corresponding monitor with this local specification. It also provides developers the ability and freedom to design their desired protocols for governing the products.

As a session-based description language, Scribble is a good subject for us to study the comparisons between it and other policy and protocol languages through discovering the atttributes and the semantical behaviours, and the user feedback on using Scribble. Also, embedding assertions (as annotations) and stateful specifications into Scribble to develop a session-based *asserted* monitoring methodology of industrial strength is one of our ongoing works.

9.2.2 Exploring Structures of Asynchronous Specifications

There are many interesting open questions in the area of verifying asynchronous specifications. Here we outline the topics that we are currently investigating.

One topic is to explore approaches to automatically translating a synchronous proper stateful specification to an asynchronous proper one. In Chapter 8, two asynchronous transformed stateful specifications Θ_{async} and Θ_{ass} are proposed for translating Θ_{sync}. However, they are proposed mostly by *intuitions* instead of *formal understandings* of Θ_{sync}. More properties are needed to figure out the algorithm for automatic translation.

Also, identifying and formalising the elements of Θ which make Θ asynchronously verifiable are important. If these elements can be identified, we can formally describe the requirements to compose an asynchronously verifiable Θ, and we can compare the

expressiveness of different asynchronously verifiable stateful specifications concretely. Some fundamental understanding about the elements in a specification can help us to analyse the relationships among Θ_{sync}, Θ_{async}, and Θ_{ass}.

As Chapter 8 proves the property of soundness characterisation (see Theorem 8.5.34), which says a commutative specification is an asynchronously verifiable specification, we plan to investigate completeness. The difficulty of completeness has been illustrated in Example 8.5.36 of Chapter 8. Also, Example 8.5.35 in Chapter 8 points out that the property of commutativity is stronger than completeness by asking the conditions hold for any possible initial value of a state because we do not know the invariants of a given specifications. To explore these invariants, the properties of structures of asynchronously verifiable specifications should be explored and stated.

Currently we aim to explore and analyse concrete forms of asynchronously verifiable specifications with different structures, informed by use cases from (129) as well as our theory, with a view to their usage in monitoring. One challenge is to find a solid asynchronously verifiable specification framework for inherently conflicting operations, such as two consecutive and overwriting updates on the same datum.

9.2.3 Sharing Memory in Stateful Asynchronous Specifications

As we discussed in Chapter 8, when a state is asserted in a predicate and its updating rules are stipulated in a specification, properly specifying the specification under asynchronous environment becomes critical. As a first step of this issue, in this thesis, we confine a state of an endpoint can only be accessed by session-roles who are obeying to the same global specification. Moreover, the state is local, which means it can only be accessed by multi-threads of the local process possessing it; it can not be shared across different domains.

One topic here is to analyse and realise sharing a state, which can only be accessed by a local process, among different sessions. In other words, although a state belongs to a particular local process, this process can join different sessions for playing different roles. With this setting, we want to investigate the structures of a specification which ensure that a state can be properly shared.

When we understand the above more complete, we hope to extend the topic to sharing memory in different sessions across different domains. For example, a state is not local to a process, but is a shared memory in the network, which can be accessed by

authenticated session-roles. This topic will explore the session-based memory sharing in cloud computing.

9.2.4 Governance with Stateful Policies

When we adopt a session-based governance framework where dynamic monitoring is based on stateful specifications, it is natural to explore the corresponding stateful policy language for specifying policies and commitments for governance.

A policy should be simply stated by a policy language so that it is readable by general users. With this principle, we are investigating expressive policy languages for policies which can specify traces which imply the changes of an endpoint field, rather than complicated stateful specifications. We have learned from Chapter 8 that it is not easy to specify an asynchronously verifiable stateful specification.

We find that there are elements which compose a specification, and those elements are the requirements should be stated in a policy. For example, if a policy $\mathfrak{tq}$ asks that "in a trace, every value carried by an output action should be unique", then Θ_{sync}, Θ_{async} and Θ_{ass} all satisfy $\mathfrak{tq}$ so that $\mathfrak{tq}$ is one of the elements which compose these specifications.

But when a policy $\mathfrak{tq}$ says "in a trace, every content carried by an output should be one increment larger than the content carried by its previous output", then only Θ_{sync} can specify, synchronously, this requirement. We say this kind of policy implies the changes of state of a local process, and it is not a proper policy under asynchronous environments.

Currently we are applying temporal logics to formalise the policy language, and wish to further analyse the properties of stateful policies with the results in Chapter 8.

APPENDIX A

Appendix for Capability-passing Calculus

A.1 Notes for the Syntax with or without Session Creation Rule

As the syntax defined in Figure 5.3 Chapter 5, we do not have a single rule for newing a session; instead, we use bound request to combine session creation and the first invitation. In (39), the rule for session creation, $\mathsf{new}\ s : G.P$, is used. The advantages of using $\mathsf{new}\ s : G.P$ are (I) clearly stating a process creates a fresh session s specified under G, (II) enjoying flexibility for representing actions. A process is able to pass a *partial* capability to another process with this design of primitives. However, this primitive also makes the representation of interleaving actions too cumbersome to analyse. For example, if the corresponding action of process $\mathsf{new}\ s : G.P$ is $\mathsf{new}\ s : G$, and T is projected from G as well as T' is projected from G', then the sequence of actions has the following six equivalent sequences of actions (represented by action labels):

$$
\begin{array}{ll}
\mathsf{new}\ s : G \cdot \mathsf{new}\ s' : G' \cdot \overline{a}\langle s[p] : T\rangle \cdot \overline{b}\langle s'[p'] : T'\rangle & \equiv \\
\mathsf{new}\ s' : G' \cdot \mathsf{new}\ s : G \cdot \overline{a}\langle s[p] : T\rangle \cdot \overline{b}\langle s'[p'] : T'\rangle & \equiv \\
\mathsf{new}\ s : G \cdot \mathsf{new}\ s' : G' \cdot \overline{b}\langle s'[p'] : T'\rangle \cdot \overline{a}\langle s[p] : T\rangle & \equiv \\
\mathsf{new}\ s' : G' \cdot \mathsf{new}\ s : G \cdot \overline{b}\langle s'[p'] : T'\rangle \cdot \overline{a}\langle s[p] : T\rangle & \equiv \\
\mathsf{new}\ s : G \cdot \overline{a}\langle s[p] : T\rangle \cdot \mathsf{new}\ s' : G' \cdot \overline{b}\langle s'[p'] : T'\rangle & \equiv \\
\mathsf{new}\ s' : G' \cdot \overline{b}\langle s'[p'] : T'\rangle \cdot \mathsf{new}\ s : G \cdot \overline{a}\langle s[p] : T\rangle &
\end{array}
$$

On the contrary, if $\overline{a}(s[p] : G)$, as an action label, is used to stand for the sequence of actions $\mathsf{new}\ s : G \cdot \overline{a}\langle s[p] : T\rangle$, there are only two equivalent sequences of actions:

$$\overline{a}(s[p] : G) \cdot \overline{b}(s'[p'] : G') \equiv \overline{b}(s'[p'] : G') \cdot \overline{a}(s[p] : G)$$

Therefore, using bound request $\overline{a}(s[p] : G)$ and free request $\overline{a}\langle s[p] : G\rangle$ separately can reduce the complexity of the analysis of the sequences of actions.

Another reason is action $\mathsf{new}\ s : G$, which corresponds to process $\mathsf{new}\ s : G.P_{\mathrm{req}}$, is always an action belonging to a requester, P_{req}, but never to an acceptor because the command of $\mathsf{new}\ s : G$ becomes duplicated when an acceptor, say P_{acc}, also has process $\mathsf{new}\ s : G.P_{\mathrm{acc}}$. Instead, adopting action notation $a(s[p] : G)$ with consistent meaning with $\overline{a}(s[p] : G)$ for declairing s is new to the process can straightforwardly prevent the duplication illustrated above. Similarly, a free accept syntax can represent the following process:

$$a(s[p] : G).b\langle s[q] : G'\rangle$$

in which there are two endpoints, a and b, are running for this process. When the first invitation from session s comes to endpoint a, it is new to the process so that $a(s[p] : G)$ is applied, while the second invitation from session s comes to endpoint b, it is not new to the process anymore so that $b\langle s[p] : G'\rangle$ is applied.

A.2 Notes for the Design of `I-` / `O`-mode Shared Names

The restrictions of creating a shared name and sending a name through interaction actions are reasoned below.

The existence of input-mode shared name is unique. Network coherence asks input-mode shared name exists whenever there is an output-mode shared name. It is defined in Definition 7.4.13 for network coherence. If we do not require this, then, as an invitor invites other parties to join a session with the knowledge of output-mode shared name, say $a : \mathtt{O}(G[p])$, which guides the invitor who is capable of playing role p specified under specification G, and guides it where the target is by a. If there is no corresponding input-mode shared name, say $a : \mathtt{im}(G[p]), \mathtt{im} \in \{\mathtt{I}, \mathtt{IO}\}$, because the monitor of an invitor is local, the monitor can not detect that there is

no receiver in the network as it approves this request, and may make the invitor wait forever for non-existent response.

Creating output-mode shared name is not allowed. As we require that whenever there is an output-mode shared name, say $a : \mathtt{O}(G[p])$, there exists its corresponding input-mode shared name $a : \mathtt{im}(G[p]), \mathtt{im} \in \{\mathtt{I}, \mathtt{IO}\}$, if we allow a process to create a new shared name with O-mode, then this requirement can be easily broken because it is possible that, when output-mode shared name is created, its corresponding input-mode shared name has not existed.

A passed shared name can only be typed with output-mode as it is received. This restriction corresponds to those above. Refer to Figure 7.4, in which the LTS of monitors is defined, the following example illustrates the issue. Assume there is a process, playing sesssion-role $s[p]$, sends a name a with specification $\mathtt{I}(G[p])$ to another process playing session-role $s[q]$. For the sender-side, a monitor can ensure whether the sender sends an input-mode shared name with specific specifications, like $s[p, q]!l_j\langle a\rangle$, a monitor can check if $\mathtt{I}(G[p]) = S_j, \Gamma \vdash S_j$ by refering to

$$s[p] : q!\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.T_i\}_{i\in I}.$$

For the receiver-side, as label l_j is chosen with an input action, for example, $s[p, q]?l_j(b)$,

$$s[q] : p?\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.T_i\}_{i\in I}$$

we can specify S_j at the label pool above as $S_j = \mathtt{I}(G[p])$; but, the receiver-side monitor cannot detect if $a = b$ because it is local, and we do not wish it takes the tasks of asking all monitors in the network one by one. If $a \neq b$, then, since a is with input-mode and it has been sent out from its original owner, there is no a with input-mode for those a with output-mode. Thus we type every received shared name with output-mode to ensure the network coherence.

A. APPENDIX FOR CAPABILITY-PASSING CALCULUS

APPENDIX B

Appendix for Dynamic Asynchronously Monitoring

B.1 Proofs for Local Safety and Transparency

For proving Theorem 7.4.3 *Local Conformance* and Theorem 7.4.5 *Local Transparency*, auxiliary definitions and lemmas are stated.

Lemma B.1.1. Let ℓ be an input action label and $\mathcal{M} \xrightarrow{\ell}$ as $\mathcal{M}$ approves label ℓ. $\mathcal{M} \xrightarrow{\ell}$ implies $\mathcal{M}[P] \xrightarrow{\ell}$.

Proof. By the definitions of LTS of $\mathcal{M}$ and $\mathcal{M}[P]$ locally, for any input action ℓ, if $\mathcal{M} \xrightarrow{\ell}$, $\mathcal{M}[P] \xrightarrow{\ell}$. ■

Theorem 7.4.3 (local safety). For each monitor $\mathcal{M}$ and local process P, we have $\mathcal{M} \models \mathcal{M}[P]$.

Proof. Let

$$\mathcal{R} = \{(\mathcal{M}_i[P_i], \mathcal{M}_i) \mid P_i \text{ and } \mathcal{M}_i \text{ are arbitrary}\} \tag{B.1}$$

We show that $\mathcal{R}$ is a conformance relation as we defined in Definition 7.4.2.

1. When ℓ is an output action, based on Definition 7.2.6, by deduction, $\mathcal{M}_0[P_0] \xrightarrow{\ell}$ implies $\mathcal{M}_0 \xrightarrow{\ell}$.
2. When ℓ is an input action, $\mathcal{M}_0 \xrightarrow{\ell}$ implies $\mathcal{M}_0[P_0] \xrightarrow{\ell}$ by Lemma B.1.1.
3. If $\mathcal{M}_0[P_0] \xrightarrow{\ell} \mathcal{M}_0'[P_0']$ and $\mathcal{M}_0 \xrightarrow{\ell} \mathcal{M}_0'$, since the relation (B.1) asks that P_0' and $\mathcal{M}_0'$ are arbitrary, we have $\mathcal{M}_0'[P_0'] \ \mathcal{R} \ \mathcal{M}_0'$ hence done.

Thus $\mathcal{R}$ is a conformance relation. ■

Definition B.1.2 (LTS of monitor-off processes (addressed processes)). We define $\mathcal{M}_1^\circ[P] \xrightarrow{\ell} \mathcal{M}_2^\circ[P']$ as the conjunction of $\mathcal{M}_1^\circ \xrightarrow{\ell} \mathcal{M}_2^\circ$ and $P \xrightarrow{l} P'$:

$$\frac{\mathcal{M}_1^\circ \xrightarrow{\ell} \mathcal{M}_2^\circ, P \xrightarrow{\ell} P'}{\mathcal{M}_1^\circ[P] \xrightarrow{\ell} \mathcal{M}_2^\circ[P']}$$

Lemma B.1.3 (determinacy of gateway transitions). If $\mathcal{M} \xrightarrow{\ell} \mathcal{M}_1$ and $\mathcal{M} \xrightarrow{\ell} \mathcal{M}_2$ then $\mathcal{M}_1 = \mathcal{M}_2$. If $\mathcal{M}^\circ \xrightarrow{\ell} \mathcal{M}_1^\circ$ and $\mathcal{M}^\circ \xrightarrow{\ell} \mathcal{M}_2^\circ$ then $\mathcal{M}_1^\circ = \mathcal{M}_2^\circ$.

Proof. By the definition of LTS of monitors, $\mathcal{M}$ is deterministic. Similarly, by the definition of LTS of $\mathcal{M}^\circ$, $\mathcal{M}^\circ$ is deterministic, thus done. ■

Definition B.1.4. We write $\mathsf{erase}(\mathcal{M}) = \mathcal{M}^\circ$ a monitor-off address obtained by erasing types, predicates and variables in $\mathcal{M}$.

Lemma B.1.5.

1. If $\mathcal{M} \xrightarrow{\ell} \mathcal{M}'$, then $\mathsf{erase}(\mathcal{M}) \xrightarrow{\ell} \mathsf{erase}(\mathcal{M}')$.
2. Given $\mathcal{M}^\circ = \mathsf{erase}(\mathcal{M})$, if $\mathcal{M} \xrightarrow{\ell} \mathcal{M}_2$ and $\mathsf{erase}(\mathcal{M}) \xrightarrow{\ell} \mathcal{M}_2^\circ$, then $\mathcal{M}_2^\circ = \mathsf{erase}(\mathcal{M}_2)$.

Proof.

1. Since $\mathcal{M}$ is deterministic, by the definition of $\mathsf{erase}(\mathcal{M})$, $\mathsf{erase}(\mathcal{M})$ is deterministic corresponding to $\mathcal{M}$. Thus if $\mathcal{M} \xrightarrow{\ell} \mathcal{M}_2$, $\mathsf{erase}(\mathcal{M}) \xrightarrow{\ell} \mathsf{erase}(\mathcal{M}_2)$.
2. Since $\mathcal{M}^\circ$ and $\mathsf{erase}(\mathcal{M})$ are deterministic, if $\mathcal{M}^\circ = \mathsf{erase}(\mathcal{M}) \xrightarrow{\ell} \mathcal{M}_2^\circ$, and together with 1. above, $\mathcal{M} \xrightarrow{\ell} \mathcal{M}_2$ implies $\mathsf{erase}(\mathcal{M}) \xrightarrow{\ell} \mathsf{erase}(\mathcal{M}_2)$, thus $\mathcal{M}_2^\circ = \mathsf{erase}(\mathcal{M}_2)$.■

Theorem 7.4.5 (local transparency). Let $\mathcal{M}^\circ = \mathsf{erase}(\mathcal{M})$. If $\mathcal{M} \models \mathcal{M}^\circ[P]$ then

$$\mathcal{M} \models \mathcal{M}^\circ[P] \sim \mathcal{M}[P].$$

Proof. By assumption, we have $\mathcal{M} \models \mathcal{M}^\circ[P]$. We also have $\mathcal{M} \models \mathcal{M}[P]$ by Theorem 7.4.3. Let $\mathcal{M}^\circ[P] = \mathcal{L}_1$ and $\mathcal{M}[P] = \mathcal{L}_2$. Let $\mathcal{R}$ be the minimum conformance relation which witnesses $\mathcal{M} \models \mathcal{L}$, i.e. $(\mathcal{L}, \mathcal{M}) \in \mathcal{R}$. Define $\mathcal{R}'$ as:

$$\mathcal{R}' = \{(\mathcal{M}, \mathcal{L}_i, \mathcal{L}_j) \mid (\mathcal{L}_i, \mathcal{M}) \in \mathcal{R}, (\mathcal{L}_j, \mathcal{M}) \in \mathcal{R}\} \tag{B.2}$$

We show $\mathcal{R}'$ is a monitored strong bisimulation for $i, j \in \{1, 2\}$, $i \neq j$.

1. Suppose $(\mathcal{M}, \mathcal{L}_1, \mathcal{L}_2) \in \mathcal{R}'$, i.e. $(\mathcal{M}, \mathcal{M}^\circ[P], \mathcal{M}[P]) \in \mathcal{R}'$ and $(\mathcal{M}^\circ[P], \mathcal{M}) \in \mathcal{R}$ and $(\mathcal{M}[P], \mathcal{M}) \in \mathcal{R}$.

   I. When ℓ is an τ or an output action, we want to prove that $\mathcal{M}^\circ[P] \xrightarrow{\ell} \mathcal{M}'^\circ[P']$ implies $\mathcal{M} \xrightarrow{\ell} \mathcal{M}'$ and $\mathcal{M}[P] \xrightarrow{\ell} \mathcal{M}'[P']$. Suppose $\mathcal{M}^\circ[P] \xrightarrow{\ell} \mathcal{M}'^\circ[P']$:

      (a) Since $\mathcal{M} \models \mathcal{M}^\circ[P]$ is given, $\mathcal{M}^\circ[P] \xrightarrow{\ell}$ implies $\mathcal{M} \xrightarrow{\ell}$ by Definition 7.4.2. Assume there exists $\mathcal{M}'$ such that $\mathcal{M} \xrightarrow{\ell} \mathcal{M}'$. Also assume that there exsits $\mathcal{M}''^\circ$ such that $\mathcal{M}^\circ[P] \xrightarrow{\ell} \mathcal{M}''^\circ[P']$, which implies $\mathcal{M}^\circ \xrightarrow{\ell} \mathcal{M}''^\circ$ and $P \xrightarrow{\ell} P'$ by deduction. Since $\mathcal{M}^\circ = \mathsf{erase}(\mathcal{M})$ is given, by Lemma B.1.5, if $\mathcal{M} \xrightarrow{\ell} \mathcal{M}'$, we have $\mathcal{M}^\circ = \mathsf{erase}(\mathcal{M}) \xrightarrow{\ell} \mathsf{erase}(\mathcal{M}')$. Since the labelled transition system of $\mathcal{M}^\circ$ is deterministic, we should have $\mathcal{M}''^\circ = \mathsf{erase}(\mathcal{M}') = \mathcal{M}'^\circ$.

      (b) With (a) above and Definition 7.2.6, $P \xrightarrow{\ell} P'$ and $\mathcal{M} \xrightarrow{\ell} \mathcal{M}'$ together induce $\mathcal{M}[P] \xrightarrow{\ell} \mathcal{M}'[P']$.

      (c) By Definition 7.4.2, we have $(\mathcal{M}'^\circ[P'], \mathcal{M}') \in \mathcal{R}$ due to $(\mathcal{M}^\circ[P], \mathcal{M}) \in \mathcal{R}$, and by Theorem 7.4.3, we have $(\mathcal{M}'[P], \mathcal{M}') \in \mathcal{R}$, so that we have $(\mathcal{M}', \mathcal{M}'^\circ[P'], \mathcal{M}'[P']) \in \mathcal{R}'$.

   II. When ℓ is an input action, suppose $\mathcal{M}^\circ[P] \xrightarrow{\ell} \mathcal{M}'^\circ[P']$ as well as $\mathcal{M} \xrightarrow{\ell} \mathcal{M}'$:

      (a) With Definition B.1.2, $\mathcal{M}^\circ[P] \xrightarrow{\ell} \mathcal{M}'^\circ[P']$ implies $P \xrightarrow{\ell} P'$ and $\mathcal{M}^\circ \xrightarrow{\ell} \mathcal{M}'^\circ$. $P \xrightarrow{\ell} P'$ and $\mathcal{M} \xrightarrow{\ell} \mathcal{M}'$ together implies $\mathcal{M}[P] \xrightarrow{\ell} \mathcal{M}'[P']$.

      (b) Since $\mathcal{M} \models \mathcal{M}^\circ[P]$ is given, with Definition 7.4.2, whenever $\mathcal{M} \xrightarrow{\ell} \mathcal{M}'$ and $\mathcal{M}^\circ[P] \xrightarrow{\ell} \mathcal{M}'^\circ[P']$, we have $(\mathcal{M}'^\circ[P'], \mathcal{M}') \in \mathcal{R}$. Together with $(\mathcal{M}'[P'], \mathcal{M}') \in \mathcal{R}$ by Theorem 7.4.3, we have $(\mathcal{M}', \mathcal{M}'^\circ[P'], \mathcal{M}'[P']) \in \mathcal{R}'$.

2. For the symmetric case, suppose $(\mathcal{M}, \mathcal{L}_2, \mathcal{L}_1) \in \mathcal{R}'$, i.e. $(\mathcal{M}, \mathcal{M}[P], \mathcal{M}^\circ[P]) \in \mathcal{R}'$ and $(\mathcal{M}[P], \mathcal{M}) \in \mathcal{R}$ and $(\mathcal{M}^\circ[P], \mathcal{M}) \in \mathcal{R}$.

   I. When ℓ is an τ or an output action, we want to prove $\mathcal{M}[P] \xrightarrow{\ell} \mathcal{M}'[P']$ implies $\mathcal{M} \xrightarrow{\ell} \mathcal{M}'$ and $\mathcal{M}^\circ[P] \xrightarrow{\ell} \mathcal{M}'^\circ[P']$. Suppose $\mathcal{M}[P] \xrightarrow{\ell} \mathcal{M}'[P']$:

      (a) $\mathcal{M}[P] \xrightarrow{\ell} \mathcal{M}'[P']$ implies $\mathcal{M} \xrightarrow{\ell} \mathcal{M}'$ as well as $P \xrightarrow{\ell} P'$.

      (b) Since $\mathcal{M}^\circ = \mathsf{erase}(\mathcal{M})$ is given, by Lemma B.1.5 and $\mathcal{M} \xrightarrow{\ell} \mathcal{M}'$ in (a), we have $\mathcal{M}^\circ \xrightarrow{\ell} \mathcal{M}'^\circ = \mathsf{erase}(\mathcal{M}')$. Together with $P \xrightarrow{\ell} P'$ in (a), we have $\mathcal{M}^\circ[P] \xrightarrow{\ell} \mathcal{M}'^\circ[P']$.

      (c) Since we have $(\mathcal{M}'^\circ[P'], \mathcal{M}') \in \mathcal{R}$ by Definition 7.4.2 and $(\mathcal{M}'[P'], \mathcal{M}') \in \mathcal{R}$ by Theorem 7.4.3, we have $(\mathcal{M}', \mathcal{M}'[P'], \mathcal{M}'^\circ[P']) \in \mathcal{R}'$.

II. When ℓ is an input action, suppose $\mathcal{M}[P] \xrightarrow{\ell} \mathcal{M}'[P']$ as well as $\mathcal{M} \xrightarrow{\ell} \mathcal{M}'$. Note that the former implies $P \xrightarrow{\ell} P'$.

   (a) Since $\mathcal{M}^\circ = \mathsf{erase}(\mathcal{M})$ is given, by Lemma B.1.5, $\mathcal{M} \xrightarrow{\ell} \mathcal{M}'$ implies $\mathsf{erase}(\mathcal{M}) \xrightarrow{\ell} \mathsf{erase}(\mathcal{M}')$ and $\mathcal{M}^\circ \xrightarrow{\ell} \mathcal{M}''^\circ$ so that $\mathcal{M}'^\circ = \mathsf{erase}(\mathcal{M}')$.

   (b) $\mathcal{M}^\circ \xrightarrow{\ell} \mathcal{M}'^\circ$ and $P \xrightarrow{\ell} P'$ imply $\mathcal{M}^\circ[P] \xrightarrow{\ell} \mathcal{M}'^\circ[P']$. Since $(\mathcal{M}'^\circ[P'], \mathcal{M}') \in \mathcal{R}$ by Definition 7.4.2, and $(\mathcal{M}'[P'], \mathcal{M}') \in \mathcal{R}$ by Theorem 7.4.3, we have $(\mathcal{M}', \mathcal{M}'[P'], \mathcal{M}'^\circ[P']) \in \mathcal{R}'$.

Thus $\mathcal{R}'$ is a monitored strong bisimulation. ■

B.2 Proofs for Global Safety

Before proving *global safety* (Theorem 7.4.16) and *global transparency* (Theorem 7.4.17), we introduce the following definitions and lemmas.

Define the set of monitored processes similarly as the one for the set of monitors defined in Definition 7.4.10.

Definition B.2.1 (a set of monitored processes). Define a set of monitored processes as

$$\prod_{i \in I} \mathcal{M}_i[P_i] = \mathcal{M}_1[P_1] \parallel \ldots \parallel \mathcal{M}_k[P_k], I = \{1, \ldots, k\}$$

Definition B.2.2 (action chain). Globally, for a sequence of actions $\ell_i{\cdot}\ell_j$, we say that ℓ_i and ℓ_j are in one *action chain* if a group of monitored processes $\prod_i \mathcal{M}_i[P_i]$ approves $\ell_i \cdot \ell_j$, denoted as

$$\prod_i \mathcal{M}_i[P_i] \xrightarrow{\ell_i \cdot \ell_j}_{\mathrm{g}},$$

then

$$\prod_i \mathcal{M}_i[P_i] \stackrel{\ell_j \cdot \ell_i}{\not\longrightarrow}_{\mathrm{g}}$$

which means ℓ_i and ℓ_j are *not* permutable.

Actions in one *action chain* are not permutable, but actions in different action chains are permutable. There could be several *action chains* in a sequence of actions. We use two examples to illustrate how an action chain works.

Example B.2.3. Assume a sequence of actions is given below

$$s'[p_1, p_2]!\langle 4\rangle \cdot \texttt{join}(s[p_3]) \cdot s'[p_1, p_2]?(4) \cdot s[p_1, p_3]!\langle \text{hi}\rangle \cdot s[p_1, p_3]?(\text{hi})$$

actions $\texttt{join}(s[p_3])$, $s[p_1, p_3]!\langle \text{hi}\rangle$ and $s[p_1, p_3]?(\text{hi})$ are in one *action chain*, we mark them as $(\texttt{join}(s[p_3]))^{\mathsf{c}_1}$, $(s[p_1, p_3]!\langle \text{hi}\rangle)^{\mathsf{c}_1}$ and $(s[p_1, p_3]?(\text{hi}))^{\mathsf{c}_1}$ to denote that they are in an action chain called c_1; actions $s'[p_1, p_2]!\langle 4\rangle, s'[p_1, p_2]?(4)$ are in another *action chain*, we mark them as $(s'[p_1, p_2]!\langle 4\rangle)^{\mathsf{c}_2}, (s'[p_1, p_2]?(4))^{\mathsf{c}_2}$ to denote that they are in an action chain called c_2. The sequence of actions can be permuted as

$$(\texttt{join}(s[p_3]))^{\mathsf{c}_1} \cdot (s'[p_1, p_2]!\langle 4\rangle)^{\mathsf{c}_2} \cdot (s[p_1, p_3]!\langle \text{hi}\rangle)^{\mathsf{c}_1} \cdot (s'[p_1, p_2]?(4))^{\mathsf{c}_2} \cdot (s[p_1, p_3]?(\text{hi}))^{\mathsf{c}_1}$$

but cannot be permuted as

$$(s'[p_1, p_2]?(4))^{\mathsf{c}_2} \cdot (s'[p_1, p_2]!\langle 4\rangle)^{\mathsf{c}_2} \cdot (s[p_1, p_3]!\langle \text{hi}\rangle)^{\mathsf{c}_1} \cdot (\texttt{join}(s[p_3]))^{\mathsf{c}_1} \cdot (s[p_1, p_3]?(\text{hi}))^{\mathsf{c}_1}$$

because $(s'[p_1, p_2]?(4))^{\mathsf{c}_2}$ cannot happen before $(s'[p_1, p_2]!\langle 4\rangle)^{\mathsf{c}_2}$, and $(s[p_1, p_3]!\langle \text{hi}\rangle)^{\mathsf{c}_1}$ cannot happen before $(\texttt{join}(s[p_3]))^{\mathsf{c}_1}$.

In the above example, there are two sessions s and s' involved. The following example shows the action chains in one session.

Example B.2.4. Given a sequence of actions

$$s[p_1, p_2]!\langle 4\rangle \cdot s[p_1, p_2]?(4) \cdot s[p_1, p_3]!\langle \text{hi}\rangle \cdot s[p_1, p_3]?(\text{hi})$$

actions $s[p_1, p_2]!\langle 4\rangle$ and $s[p_1, p_2]?(4)$ are in one *action chain*, we mark them as $(s[p_1, p_2]!\langle 4\rangle)^{\mathsf{c}_1}$, $(s[p_1, p_2]?(4))^{\mathsf{c}_1}$ to denote that they are in an action chain called c_1; actions $s[p_1, p_3]!\langle \text{hi}\rangle$ and $s[p_1, p_3]?(\text{hi})$ are in another *action chain*, we mark them as $(s[p_1, p_3]!\langle \text{hi}\rangle)^{\mathsf{c}_2}, (s[p_1, p_3]?(\text{hi}))^{\mathsf{c}_2}$ to denote that they are in an action chain called c_2. The sequence of actions can be permuted as

$$(s[p_1, p_2]!\langle 4\rangle)^{\mathsf{c}_1} \cdot (s[p_1, p_3]!\langle \text{hi}\rangle)^{\mathsf{c}_2} \cdot (s[p_1, p_2]?(4))^{\mathsf{c}_1} \cdot (s[p_1, p_3]?(\text{hi}))^{\mathsf{c}_2}$$

but cannot be permuted as

$$(s[p_1, p_2]?(4))^{\mathsf{c}_1} \cdot (s[p_1, p_3]!\langle \text{hi}\rangle)^{\mathsf{c}_2} \cdot (s[p_1, p_3]?(\text{hi}))^{\mathsf{c}_2} \cdot (s[p_1, p_2]!\langle 4\rangle)^{\mathsf{c}_1}$$

because $(s[p_1, p_2]?(4))^{\mathsf{c}_1}$ cannot happen before $(s[p_1, p_2]!\langle 4\rangle)^{\mathsf{c}_1}$.

Definition B.2.5 (input action chain). An *input action chain* is a subset of an action chain. We denote an action ℓ, which is either $s[p_1, p_2]?l(v)$, or $a(s[p] : G)$, or $\texttt{join}(s[p])$, in an *input action chain* c as $(\ell)^{\mathsf{c}_{in}}$.

Lemma B.2.6. Let N be a monitored network and $\prod_i \mathcal{M}_i[P_i] \in N$. Assume an input action chain $\mathsf{c} = (\ell_0)^{\mathsf{c}_{in}} \cdot (\ell_1)^{\mathsf{c}_{in}} \cdot \ldots \cdot (\ell_n)^{\mathsf{c}_{in}}$ is given and

$$\prod_i \mathcal{M}_i[P_i] \xrightarrow{(\ell_0)^{\mathsf{c}_{in}} \cdot (\ell_1)^{\mathsf{c}_{in}} \cdot \ldots \cdot (\ell_n)^{\mathsf{c}_{in}}}_{\mathrm{g}}$$

if $\prod_i \mathcal{M}_i[P_i] \xrightarrow{(\ell_0)^{\mathsf{c}_{in}}}_{\mathrm{g}} \prod_i \mathcal{M}'_i[P'_i]$, then $\prod_i \mathcal{M}'_i[P'_i] \xrightarrow{(\ell_1)^{\mathsf{c}_{in}} \cdot (\ell_2)^{\mathsf{c}_{in}} \cdot \ldots \cdot (\ell_n)^{\mathsf{c}_{in}}}_{\mathrm{g}}$.

Proof. Let $\mathsf{c} = (\ell_0)^{\mathsf{c}_{in}} \cdot (\ell_1)^{\mathsf{c}_{in}} \cdot \ldots \cdot (\ell_n)^{\mathsf{c}_{in}}$ be an input action chain.

$$\prod_i \mathcal{M}_i[P_i] \xrightarrow{(\ell_0)^{\mathsf{c}_{in}} \cdot (\ell_1)^{\mathsf{c}_{in}} \cdot \ldots \cdot (\ell_n)^{\mathsf{c}_{in}}}_{\mathrm{g}}$$

means that the pending messages corresponding to $(\ell_0)^{\mathsf{c}_{in}} \cdot (\ell_1)^{\mathsf{c}_{in}} \cdot \ldots \cdot (\ell_n)^{\mathsf{c}_{in}}$ can be received by the group of monitored processes $\prod_i \mathcal{M}_i[P_i]$:

$$\prod_i \mathcal{M}_i[P_i] \xrightarrow{(\ell_0)^{\mathsf{c}_{in}}}_{\mathrm{g}} \prod_i \mathcal{M}'_i[P'_i] \xrightarrow{(\ell_1)^{\mathsf{c}_{in}}}_{\mathrm{g}} \prod_i \mathcal{M}''_i[P''_i] \xrightarrow{(\ell_2)^{\mathsf{c}_{in}}}_{\mathrm{g}} \ldots\ldots \prod_i \mathcal{M}^*_i[P^*_i]$$

If $\prod_i \mathcal{M}_i[P_i] \xrightarrow{(\ell_0)^{\mathsf{c}_{in}}}_{\mathrm{g}} \prod_i \mathcal{M}'_i[P'_i]$, the input action chain becomes $\mathsf{c}' = (\ell_1)^{\mathsf{c}_{in}}, .., (\ell_n)^{\mathsf{c}_{in}}$, where $(\ell_0)^{\mathsf{c}_{in}}$ is received, and $(\ell_1)^{\mathsf{c}_{in}}$ becomes the first action in c':

$$\prod_i \mathcal{M}'_i[P'_i] \xrightarrow{(\ell_1)^{\mathsf{c}_{in}}}_{\mathrm{g}} \prod_i \mathcal{M}''_i[P''_i] \xrightarrow{(\ell_2)^{\mathsf{c}_{in}}}_{\mathrm{g}} \ldots\ldots \prod_i \mathcal{M}^*_i[P^*_i]$$

Thus $\prod_i \mathcal{M}'_i[P'_i] \xrightarrow{(\ell_1)^{\mathsf{c}_{in}} \cdot \ldots \cdot (\ell_n)^{\mathsf{c}_{in}}}_{\mathrm{g}}$. ■

Lemma 7.4.7 (receivability). Let N be a monitored network and ℓ be an input action. If N is receivable and $N \xrightarrow{\ell}_{\mathrm{g}} N'$, then N' is receivable.

Proof. Assume $\{\mathcal{M}_i[P_i]\}_{i \in I} \in N$. That N is receivable means every pending message floating in the global queue can be absorbed by some monitored process in N. Thus, as N is receivable, there exists a sequence of actions, say $\vec{\ell}$, consisting of join and those

input actions corresponding to *all* pending messages in the network N, such that

$$\prod_i \mathcal{M}_i[P_i] \xrightarrow{\vec{\ell}}_{\mathrm{g}} \prod_i \mathcal{M}_i^*[P_i^*],$$

all the messages are absorbed by some processes in $\prod_i \mathcal{M}_i[P_i]$. Assume the corresponding set of *input action chains* of $\vec{\ell}$ is $\{\mathsf{c}_{in_j}\}_{j\in J}$, which means that the actions in $\vec{\ell}$ can be partitioned into several input action chains. We mark each action $\ell \in \vec{\ell}$ as $(\ell)^{\mathsf{c}_{in_j}}$ to denote that action ℓ belongs to input action chain c_{in_j}.

We rewrite the sequence of actions in (B.2) as

$$\vec{\ell} \equiv (\mathfrak{L})^{\mathsf{c}_{in_0}} \ldots (\mathfrak{L})^{\mathsf{c}_{in_j}} \ldots (\mathfrak{L})^{\mathsf{c}_{in_J}}$$

by marking each action and collecting actions belonging to the same input action chain together, where $(\mathfrak{L})^{\mathsf{c}_{in_j}} = (\ell_{j_0})^{\mathsf{c}_{in_j}} \cdot (\ell_{j_1})^{\mathsf{c}_{in_j}} \ldots (\ell_{j_n})^{\mathsf{c}_{in_j}}$.

If

$$\prod_i \mathcal{M}_i[P_i] \xrightarrow{\ell}_{\mathrm{g}} \prod_i \mathcal{M}'_i[P'_i] \in N',$$

there exists an input action chain c_{in_k} such that $\ell \in (\mathfrak{L})^{\mathsf{c}_{in_k}} = (\ell_{k_0})^{\mathsf{c}_{in_k}} (\ell_{k_1})^{\mathsf{c}_{in_k}} \ldots (\ell_{k_m})^{\mathsf{c}_{in_k}}$; we thus mark ℓ as $(\ell)^{\mathsf{c}_{in_k}}$. By Definition B.2.5, no action in the action chain is permutable, action $(\ell)^{\mathsf{c}_{in_k}}$ should be the first action in chain c_{in_k}, thus $(\ell)^{\mathsf{c}_{in_k}} = (\ell_{k_0})^{\mathsf{c}_{in_k}}$.

By (B.2), that action $(\ell_{k_0})^{\mathsf{c}_{in_k}}$ is removed from $(\mathfrak{L})^{\mathsf{c}_{in_0}} \ldots (\mathfrak{L})^{\mathsf{c}_{in_j}} \ldots (\mathfrak{L})^{\mathsf{c}_{in_k}} \ldots (\mathfrak{L})^{\mathsf{c}_{in_J}}$ in (B.2), which becomes $(\mathfrak{L})^{\mathsf{c}_{in_0}} \ldots (\mathfrak{L})^{\mathsf{c}_{in_j}} \ldots (\mathfrak{L})^{\mathsf{c}_{in_{k'}}} \ldots (\mathfrak{L})^{\mathsf{c}_{in_J}}$. Note that $(\mathfrak{L})^{\mathsf{c}_{in_{k'}}} = (\ell_{k_1})^{\mathsf{c}_{in_k}} (\ell_{k_2})^{\mathsf{c}_{in_k}} \ldots (\ell_{k_m})^{\mathsf{c}_{in_k}}$.

By Lemma B.2.6,

$$\prod_i \mathcal{M}'_i[P'_i] \xrightarrow{(\mathfrak{L})^{\mathsf{c}_{in_0}} \ldots (\mathfrak{L})^{\mathsf{c}_{in_j}} \ldots (\mathfrak{L})^{\mathsf{c}_{in_{k'}}} \ldots (\mathfrak{L})^{\mathsf{c}_{in_J}}}_{\mathrm{g}} .$$

By Definition 7.4.6, N' is receivable. ■

The definitions of consisteny of group of monitors and monitored network coherence are reminded below:

Definition 7.4.12 (consistency of a group of monitors). Let $\mathbb{M}$ be a group of monitors, and $s[p]^\circ$ stand for either $s[p]$ or $s[p]^\bullet$. Let $\forall \mathcal{M}_i \in \mathbb{M}$, $\mathcal{M}_i = \Gamma_i, \Delta_i$. If all of the following conditions hold,

1. (a) $\forall \mathcal{M}_i \in \mathbb{M}$, if $a : \mathtt{im}(G[p]) \in \mathcal{M}_i$, then there is no other $a : \mathtt{I}(G[p]) \in \mathcal{M}_j$ or $a : \mathtt{IO}(G[p]) \in \mathcal{M}_j$, $\mathcal{M}_j \in \mathbb{M}$.

   (b) $\forall \mathcal{M}_i \in \mathbb{M}$, if $\forall s, s[p]^\circ : T \in \mathcal{M}_i$, then $s[p]^\circ \notin \mathcal{M}_j$, $j \neq i$, $\mathcal{M}_j \in \mathbb{M}$.

2. If $a : \mathtt{O}(G[p]) \in \mathcal{M}_i$, then there exists $\mathcal{M}_j$ such that $a : \mathtt{im}(G[p]) \in \mathcal{M}_j$.

3. Assume $p_1 \neq p_2$. As $\Delta_i(s[p_1]^\circ) = T$ and

$$p_2!\{l_k(x_k{:}S_k)\langle\langle A_k\rangle\rangle.G_k \upharpoonright p_2\}_{k\in I} \subseteq T,$$

   there exists j such that $\Delta_j(s[p_2]^\circ) = T'$ and

$$p_1?\{l_k(x_k{:}S_k)\langle\langle A'_k\rangle\rangle.G_k \upharpoonright p_1\}_{k\in I} \subseteq T'$$

   with $A_k\{v/x_k\} \downarrow \mathsf{true}$ iff $A'_k\{v/x_k\} \downarrow \mathsf{true}$.

4. $\bigcup_i \Delta_i = \bigcup_{1\leq j\leq n}\{s_j[p_{j1}] : T_{j1}, .., s_j[p_{jk}] : T_{jk}, .., s_j[p_{jm_j}] : T_{jm_j}\}$ where, for each s_j, there is G_j such that $G_j \upharpoonright p_{jk} = T_{jk}$ for $1 \leq k \leq m_j$.

then the group of monitors, $\mathbb{M}$, is consistent.

Definition 7.4.13 (monitored network coherence). N is *coherent* if all of its pending messages are *receivable* up to *permutation* of actions $\curvearrowright$ and, after these messages have been received, the resulting group of monitors which guard all local process in N, say $\mathbb{M}$, is consistent.

Before proving the theorem of global safety, we use the following lemma to show that as a network is coherent and an output action is ready to fire, then its corresponding input is ready for it.

Lemma B.2.7. Assume network N is coherent with its consistent group of monitors, say $\prod_i \mathcal{M}_i$, where $\mathcal{M}_i = \Gamma_i, \Delta_i$, and $\exists \mathcal{M}_s \in \prod_i \mathcal{M}_i$ such that $\Delta_s(s[p]^\bullet) \curvearrowright q!\{l_k(x_k : S_k)\langle\langle A_k\rangle\rangle.T_k\}_{k\in I}$, then $\exists \mathcal{M}_r \in \prod_i \mathcal{M}_i$ such that $\Delta_r(s[q]^\bullet) \curvearrowright p?\{l_k(x_k : S_k)\langle\langle A'_k\rangle\rangle.T'_k\}_{k\in I}$ with $A_k\{v/x_k\} \downarrow \mathsf{true}$ iff $A'_k\{v/x_k\} \downarrow \mathsf{true}$.

Proof. We prove it by contradiction. Since $\Delta_s(s[p]^\bullet) \curvearrowright q!\{l_k(x_k : S_k)\langle\langle A_k\rangle\rangle.T_k\}_{k\in I}$, we have $q!\{l_k(x_k : S_k)\langle\langle A_k\rangle\rangle.T_k\}_{k\in I} \subseteq \Delta_s(s[p]^\bullet)$. And since N is coherent, there exists

some monitor, say $\mathcal{M}_r \in \prod_i \mathcal{M}_i$, such that $p?\{l_k(x_k : S_k)\langle\langle A'_k\rangle\rangle.T''_k\}_{k\in I} \subseteq \Delta_r(s[q]^\bullet)$ by rule 3. in Definition 7.4.12. If

$$\Delta_r(s[q]^\bullet) \not\curvearrowright p?\{l_k(x_k : S_k)\langle\langle A'_k\rangle\rangle.T'_k\}_{k\in I},$$

(note that T'_k and T''_k) based on Definition 6.6.8, it must be an action with the same subject suppressing the permutation:

case (a) $\Delta_r(s[q]^\bullet) \curvearrowright p?\{l'_j(x'_j : S'_j)\langle\langle B_j\rangle\rangle.p?\{l_k(x_k : S_k)\langle\langle A'_k\rangle\rangle.T''_k\}_{k\in I}\}_{j\in J}$, or

case (b) $\Delta_r(s[q]^\bullet) \curvearrowright p!\{l'_j(x'_j : S'_j)\langle\langle B_j\rangle\rangle.p?\{l_k(x_k : S_k)\langle\langle A'_k\rangle\rangle.T''_k\}_{k\in I}\}_{j\in J}$

Case (a) implies that, for specification $\Delta_s(s[p]^\bullet)$, there is another action $q!$ positioning before $q!$ such that

$$\Delta_s(s[p]^\bullet) \curvearrowright q!\{l'_j(x'_j : S'_j)\langle\langle B'_j\rangle\rangle.q!\{l_k(x_k : S_k)\langle\langle A_k\rangle\rangle.T'''_k\}_{k\in I}\}_{j\in J}.$$

Similarly, case (b) implies that, for specification $\Delta_s(s[p]^\bullet)$, there is an action $q?$ positioning before $q!$ such that

$$\Delta_s(s[p]^\bullet) \curvearrowright q?\{l'_j(x'_j : S'_j)\langle\langle B'_j\rangle\rangle.q!\{l_k(x_k : S_k)\langle\langle A_k\rangle\rangle.T'''_k\}_{k\in I}\}_{j\in J}.$$

Both cases lead to $\Delta_s(s[p]^\bullet) \not\curvearrowright q!\{l_k(x_k : S_k)\langle\langle A_k\rangle\rangle.T_k\}_{k\in I}$. Therefore, by contradiction, we prove the statement is true. ■

Lemma B.2.8. Let $s[p]^\circ$ stand for either $s[p]$ or $s[p]^\bullet$. Assume $\forall \mathcal{M}_i \in N, \forall s,\ s[p]^\circ : T \in \mathcal{M}_i$ implies $s[p]^\circ \notin \mathcal{M}_j$, $j \neq i$, $\mathcal{M}_j \in N$. If $N \xrightarrow{\ell}_{\mathrm{g}} N'$ and rule 1.(a) in Definition 7.4.12 holds for N and N', we have $\forall \mathcal{M}'_k \in N', \forall s,\ s[p]^\circ : T \in \mathcal{M}'_k$ implies $s[p]^\circ \notin \mathcal{M}'_j$, $j \neq k$, $\mathcal{M}'_j \in N$.

Proof. As explained in Lemma 7.4.15, as rule 1.(a) in Definition 7.4.12 holds, $a : \mathtt{im}(G[p])$ is unique to the network. Only three rules, [req-b] and [acc-b/f], may affect the uniqueness of $s[p]^\circ : T$ as $N \xrightarrow{\ell}_{\mathrm{g}} N'$. When $\ell = \overline{a}(s[p_j] : G)$, there exists $\mathcal{M}_s \in N$ such that $\mathcal{M}_s \xrightarrow{\overline{a}(s[p_j]:G)} \mathcal{M}'_s$ implying s is new to $\mathcal{M}_s$ and $\{s[p_k] : G \upharpoonright p_k\}_{k\in I\setminus\{j\}} \in \mathcal{M}'_s$, where each $s[p_k] : G \upharpoonright p_k$ only exists in $\mathcal{M}'_s$ because s is unique to the network (by Lemma 7.4.15), and every role defined in a global specification is unique (Definition 6.2.3); while $s[p_j] : G \upharpoonright p_j$, which is sent as an invitation to the endpoint having $a : \mathtt{im}(G[p_j])$, say it is the endpoint guarded by monitor $\mathcal{M}_r$. As $\mathcal{M}_r$ accepts this invitation by approving action $a(s[p_j] : G)$ or $a\langle s[p_j] : G\rangle$, $s[p_j] : G \upharpoonright p_j$ is added into $\mathcal{M}_r$, and it only exists in $\mathcal{M}_r$ network-wise because $a : \mathtt{im}(G[p_j])$ only exists in $\mathcal{M}_r$: Only a monitor having

$a : \mathtt{im}(G[p_j])$ can accept the capability of $s[p_j] : G \upharpoonright p_j$. With the same reasoning, when $\ell = a(s[p_j] : G)$ or $\ell = a\langle s[p_j] : G\rangle$ is approved by some monitor, say $\mathcal{M}_k$, then $s[p_j] : G \upharpoonright p_j \in \mathcal{M}_k$, $s[p_j] : G \upharpoonright p_j$ only exists in $\mathcal{M}_k$ because only a monitor having $a : \mathtt{im}(G[p_j])$ can accept the capability of $s[p_j] : G \upharpoonright p_j$. In conclusion, no action affects the uniqueness of $s[p]^\circ : T$ to the network as $N \xrightarrow{\ell}_{\mathrm{g}} N'$, thus N' holds the statement.■

Theorem 7.4.16 (global safety). Let N be coherent. Then $N \xrightarrow{\ell}_{\mathrm{g}} N'$ implies N' is coherent.

Proof. Assume N is coherent so that the group of monitors, say $\mathbb{M}$, guarding every local process in N is consistent. Assume $\prod_i \mathcal{M}_i[P_i] \in N$, where $\prod_i \mathcal{M}_i[P_i]$ is the composition of monitored processes in N. Let $\mathcal{M}_i = \Gamma_i, \Delta_i$ and $\mathtt{im} \in \{\mathtt{I}, \mathtt{IO}\}$. For convenience, we use $\prod_i \mathcal{M}_i$ to denote the group of monitors corresponding to the group of monitored processes $\prod_i \mathcal{M}_i[P_i]$, and use + to denote adding (or having) elements in Γ_i or Δ_i. Note that, by Lemma 7.4.7, if N is receivable and $N \xrightarrow{\ell}_{\mathrm{g}} N'$ where ℓ is an *input*, then N' is receivable. Also note that, by Lemma B.2.8, rule 1.(b) of Definition 7.4.12 is always true when N is consistent and $N \xrightarrow{\ell}_{\mathrm{g}} N'$ for any ℓ, thus we only need to prove rule 1.(a). If $N \xrightarrow{\ell}_{\mathrm{g}} N'$, we want to prove the following cases of actions still make the resulting group of monitors in N' consistent. Note that the rules 1.(a), 2., 3., and 4. are referring to the rules defined in Definition 7.4.12.

case (shared name) - If ℓ is $\mathsf{new}\ a : \mathtt{im}(T[p])$, then rules 2., 3. and 4. of the consistency of group of monitors guarding all local processes in N are not affected. We analyse the effects for rule 1.(a). There is some $\mathcal{M}_j[P] \in \prod_i \mathcal{M}_i[P_i]$ and $\mathcal{M}_j[P] \xrightarrow{\mathsf{new}\ a:\mathtt{im}(G[p])} \mathcal{M}'_j[P']$ such that $N \xrightarrow{\mathsf{new}\ a:\mathtt{im}(G[p])}_{\mathrm{g}} N'$; it implies that, when $\mathtt{im} = \mathtt{I}$, there is no $a : \mathtt{I}(G[p])$ nor $a : \mathtt{IO}(G[p])$ in $\mathcal{M}_j$ so that $\mathcal{M}_j$ allows this action; similarly, when $\mathtt{im} = \mathtt{IO}$, there is no $a : \mathtt{I}(G[p])$ nor $a : \mathtt{IO}(G[p])$ in $\mathcal{M}_j$. Therefore, for $\Gamma_j \in \mathcal{M}_j$ and $\Gamma'_j \in \mathcal{M}'_j$, Γ'_j is the configuration resulting from $\Gamma_j + a : \mathtt{im}(G[p])$, which induces $\Gamma'_j = \Gamma_j, a : \mathtt{im}(G[p])$, $a : \mathtt{im}(G[p]) \notin \Gamma_j$. Thus rule 1.(a) is satisfied.

case (bound request) - If ℓ is $\overline{a}(s[p_k] : G)$, there is some $\mathcal{M}_j[P_j] \in \prod_i \mathcal{M}_i[P_i]$ and $\mathcal{M}_j[P_j] \xrightarrow{\overline{a}(s[p_k]:G)} \mathcal{M}'_j[P'_j]$ such that $N \xrightarrow{\overline{a}(s[p_k]:G)}_{\mathrm{g}} N' = (\nu s)(N'_1 \parallel H \cdot \overline{a}\langle s[p] : G\rangle)$. This action combines two parts: creating a fresh session and requesting through sending an invitation. As for the part of newing a session, as a session is newed, $\{s[p_i] : G \upharpoonright p_i\}_{i \in I \setminus \{k\}}$ are added into Δ_j, which becomes $\Delta'_j = \Delta_j + \{s[p_i] : G \upharpoonright p_i\}_{i \in I \setminus \{k\}}$. No other $\Delta_i, i \neq j$ is affected by newing a session, thus rules 1.(a), 2., and 3. are hold for newing a session. For rule 4., since N is coherent and G is approved by monitor $\mathcal{M}_j$,

G is well-formed, thus the group of monitors guarding all local processes in N' satisfies rule 4. immediately.

As for the part of requesting an invitation to endpoint a, the reasoning is as same as that for **case (free request)** below.

case (free request) - If ℓ is $\overline{a}\langle s[p] : G\rangle$, for receivability, there is some $\mathcal{M}_s[P_s] \in \prod_i \mathcal{M}_i[P_i]$ and $\mathcal{M}_s[P_s] \xrightarrow{\overline{a}\langle s[p]:G\rangle} \mathcal{M}'_s[P'_s]$ such that $N_1 \parallel H \equiv N \xrightarrow{\overline{a}\langle s[p]:G\rangle}_{\mathrm{g}} N' = N'_1 \parallel H \cdot \overline{a}\langle s[p] : G\rangle$; it implies that, there exists $\mathcal{M}_s \in N$, in which it has $\Gamma_s(a) = \mathtt{O}(G[p])$. Since N is coherent, according to rule 2., there must exist some $\mathcal{M}_r, \mathcal{M}_r[P_r] \in N$ such that it has $\Gamma_r(a) = \mathtt{im}(G[p])$, implying $\mathcal{M}_r[P_r]$ is able to receive invitation $\overline{a}\langle s[p] : G\rangle$. Since action $\overline{a}\langle s[p] : G\rangle$ does not affect the existence of $a : \mathtt{im}(G[p]) \in \Gamma_r$, $a : \mathtt{im}(G[p])$ still exists in the resulting network $N' \equiv N'_1 \parallel H \cdot \overline{a}\langle s[p] : G\rangle$. Thus N' is able to receive the additional message $\overline{a}\langle s[p] : G\rangle$, and messages $H \cdot \overline{a}\langle s[p] : G\rangle$ are receivable to N'. Below we discuss if N' satisfies rules 1.(a), 2., 3. and 4. after receiving $\overline{a}\langle s[p] : G\rangle$.

$N \xrightarrow{\overline{a}\langle s[p]:G\rangle}_{\mathrm{g}} N'$ and N' is receivable, by rules $\lfloor$MG-REQ-F-OUT$\rfloor$ and $\lfloor$MG-ACC-B/F-IN$\rfloor$, together imply $\exists\, \mathcal{M}_s[P_s], \mathcal{M}_r[P_r] \in N$ such that

$$N = N_0 \parallel \mathcal{M}_s[P_s] \parallel \mathcal{M}_r[P_r] \parallel H \xrightarrow{\overline{a}\langle s[p]:G\rangle}_{\mathrm{g}} N_0 \parallel \mathcal{M}'_s[P'_s] \parallel \mathcal{M}_r[P_r] \parallel H \cdot \overline{a}\langle s[p] : G\rangle = N'$$

where $a : \mathtt{O}(G[p]) \in \Gamma_s$, and

$$N' = N_0 \parallel \mathcal{M}'_s[P'_s] \parallel \mathcal{M}_r[P_r] \parallel H \cdot \overline{a}\langle s[p] : G\rangle \xrightarrow{\ell'}_{\mathrm{g}} N_0 \parallel \mathcal{M}'_s[P'_s] \parallel \mathcal{M}'_r[P'_r] \parallel H$$

ℓ' is $a(s[p] : G)$ or $a\langle s[p] : G\rangle$, where $a : \mathtt{im}(G[p]) \in \Gamma_r$. The invitation $\overline{a}\langle s[p] : G\rangle$ in $H \cdot \overline{a}\langle s[p] : G\rangle$ can be absorbed to the receiver when its monitor approves it. By rules [REQ-F] and [ACC-B/F], no a is added into / deleted from Γ_s or Γ_r, and $a : \mathtt{O}(G[p]) \in \Gamma_s$ and $a : \mathtt{im}(G[p]) \in \Gamma_r$ are respectively still in Γ_s and Γ_r, and their configurations do not change, thus hold rules 1.(a) and 2.

Let $\dagger \in \{!, ?\}$ and when $\dagger = ?$, let $\overline{\dagger} = !$; when $\dagger = !$, let $\overline{\dagger} = ?$. Assume in $\mathcal{M}_s$, $\Delta_s(s[p]^\circ) = T \curvearrowright q \dagger \{l_k(x_k : S_k)\langle\!\langle A_k\rangle\!\rangle.G_k \upharpoonright p\}_{k\in I}$. Since N is coherent, by Lemma B.2.7, there exists $\mathcal{M}_m[P_m]$ who has $\Delta_m(s[q]^\circ) \curvearrowright p\overline{\dagger}\{l_k(x_k : S_k)\langle\!\langle A'_k\rangle\!\rangle.G_k \upharpoonright q\}_{k\in I}$ with $\forall v, \exists\, k \in I, A_k\{v/x_k\} \downarrow \mathsf{true}$ iff $A'_k\{v/x_k\} \downarrow \mathsf{true}$.

1. For rule 3., since N is coherent, $s[p]^\circ$ only exists in Δ_s. When $s[p]^\circ$ is removed from Δ_s and added into Δ_r, by Lemma B.2.8, $s[p]^\circ$ is still unique in N' and only exists in Δ_r. Moreover, there still exists $\mathcal{M}_m[P_m]$ whose $\Delta_m(s[q]^\circ) \curvearrowright p\overline{\dagger}\{l_k(x_k : S_k)\langle\!\langle A'_k\rangle\!\rangle.G_k \upharpoonright q\}_{k\in I}$ being ready for the counter action going to/coming from

$s[p]^\circ$.

2. For rule 4., since N is coherent, $s[p]^\circ = G \upharpoonright p$, which still holds when $s[p]^\circ$ is removed from Δ_s and added into Δ_r by rules [REQ-F] and [ACC-B/F]. Thus the resulting monitors still satisfy rule 4.

case (output) - if ℓ is an output action $s[p_1, p_2]!l\langle v\rangle$, assume N approves this action:

$$N \equiv N_1 \parallel H \xrightarrow{s[p_1,p_2]!l\langle v\rangle}_{\mathrm{g}} N_1' \parallel H \cdot s\langle p_1, p_2, l\langle v\rangle\rangle \equiv N', \tag{B.3}$$

where H can be empty. Equation (B.3) infers that $\langle p_1, p_2, l\langle v\rangle\rangle$ is the only *additional* pending message to the original network N.

In the follows, we first prove that $H \cdot s\langle p_1, p_2, l\langle v\rangle\rangle$ is receivable to N'. Equation (B.3) indicates that $\exists\ \mathcal{M}_s \in N$, $\Delta_s(s[p_1]^\bullet) \curvearrowright p_2!\{l_k(x_k : S_k)\langle\langle A_k\rangle\rangle.T_k\}_{k\in I}$ approves action $\ell = s[p_1, p_2]!l\langle v\rangle$, and, due to coherence and Lemma B.2.7, implies $\exists \mathcal{M}_r \in N$ such that $\Delta_r(s[p_2]^\bullet) \curvearrowright p_1?\{l_k(x_k : S_k)\langle\langle A_k'\rangle\rangle.T_k'\}_{k\in I}$ with $\forall v, \exists\ k \in I, A_k\{v/x_k\} \downarrow \mathsf{true}$ iff $A_k'\{v/x_k\} \downarrow \mathsf{true}$. Assume $\prod_i \mathcal{M}_i$ is the group of monitors of monitored processes $\prod_i \mathcal{M}_i[P_i] \in N$, which approves to receive all pending messages in H.

1. If $\mathcal{M}_s \notin \prod_i \mathcal{M}_i$, any monitor in $\prod_i \mathcal{M}_i$ remains the same after ℓ happens. For the messages in H, they are receivable to N' (according to $\prod_i \mathcal{M}_i$). As for message $s\langle p_1, p_2, l\langle v\rangle\rangle$, since action $s[p_1, p_2]!l\langle v\rangle$ can fire in N:

   (a) If $\mathcal{M}_r = \mathcal{M}_s$, since action $s[p_1, p_2]!l\langle v\rangle$ does not affect the configuration of $\Delta_s(s[p_2]^\bullet) = \Delta_r(s[p_2]^\bullet)$, $s\langle p_1, p_2, l\langle v\rangle\rangle$ is receivable according to $\mathcal{M}_r$.

   (b) If $\mathcal{M}_r \in \prod_i \mathcal{M}_i$, by Lemma B.2.7, the receiving action $p_1?$ in $\Delta_r(s[p_2]^\bullet)$ is permutable to other receiving actions corresponding to messages in H, so that $\mathcal{M}_r$ can receive messages in H and $s\langle p_1, p_2, l\langle v\rangle\rangle$ in any order.

   (c) If $\mathcal{M}_r \notin (\prod_i \mathcal{M}_i \cup \mathcal{M}_s)$, then it is trivial that $s\langle p_1, p_2, l\langle v\rangle\rangle$ is receivable to $\mathcal{M}_r$.

2. If $\mathcal{M}_s \in \prod_i \mathcal{M}_i$, let $N \equiv N_0 \parallel \prod_i \mathcal{M}_i[P_i] \parallel H$, $N \xrightarrow{s[p_1,p_2]!l\langle v\rangle}_{\mathrm{g}} N_0 \parallel \prod_{i\neq s} \mathcal{M}_i[P_i] \parallel \mathcal{M}_s'[P_s'] \parallel H \cdot s\langle p_1, p_2, l\langle v\rangle\rangle$. As for those in H receivable to $\prod_{i\neq s} \mathcal{M}_i$, straightforwardly, they are still receivable to $\prod_{i\neq s} \mathcal{M}_i$ after action $s[p_1, p_2]!l\langle v\rangle$ fires. As for those in H receivable to $\mathcal{M}_s$ but not received at $\Delta_s(s[p_1]^\bullet)$, because action $s[p_1, p_2]!l\langle v\rangle$ has no effect at the configurations of other session-roles, they are still receivable to $\mathcal{M}_s$ at other session-roles. As for those in H but receivable at $\Delta_s(s[p_1]^\bullet)$, the messages should be in the form: $s\langle p, p_1, l'\langle v'\rangle\rangle$. Since

$\Delta_s(s[p_1]^\bullet) \curvearrowright p_2!\{l_k(x_k : S_k)\langle\!\langle A_k \rangle\!\rangle.T_k\}_{k\in I}$, by Definition 6.6.8, the configuration of $\Delta_s(s[p_1]^\bullet)$ should be

$$\Delta_s(s[p_1]^\bullet) \curvearrowright p_2!\{l_k(x_k : S_k)\langle\!\langle A_k \rangle\!\rangle.p?\{l'_j(x'_j : S'_j)\langle\!\langle B_j \rangle\!\rangle.T'_j\}_{j\in I}\}_{k\in I}$$

which shows that those messages are still receivable at $\Delta_s(s[p_1]^\bullet)$ after action $s[p_1, p_2]!l\langle v\rangle$ fires. As for message $s\langle p_1, p_2, l\langle v\rangle\rangle$, the reasonings are as same as those above.

Thus the messages in $H \cdot s\langle p_1, p_2, l\langle v\rangle\rangle$ are receivable to network N'.

Then we show that, after receiving all messages in $H \cdot s\langle p_1, p_2, l\langle v\rangle\rangle$, the resulting monitors in N' are consistent. Since rules of 1.(a) and 2. for the consistency of monitors (see Definition 7.4.12) are not affected by action $s[p_1, p_2]!\langle v\rangle$, only rules 3. and 4. need discussions. With the analyses above, since N is consistent, $\exists \mathcal{M}_s, \mathcal{M}_r \in N, \exists G$ is well-formed and $G = p_1 \to p_2 : \{l_k(x_k : S_k)\langle\!\langle A_k \rangle\!\rangle.G_k\}_{k\in I}$ such that

$$\begin{array}{lcl} \Delta_s(s[p_1]^\bullet) & \curvearrowright & G \upharpoonright p_1 = p_2!\{l_k(x_k : S_k)\langle\!\langle A_k \rangle\!\rangle.G_k \upharpoonright p_1\}_{k\in I} \\ \Delta_r(s[p_2]^\bullet) & \curvearrowright & G \upharpoonright p_2 = p_1?\{l_k(x_k : S_k)\langle\!\langle A'_k \rangle\!\rangle.G_k \upharpoonright p_2\}_{k\in I} \end{array}$$

with $\forall v, \exists\, k \in I, A_k\{v/x_k\} \downarrow \mathsf{true}$ iff $A'_k\{v/x_k\} \downarrow \mathsf{true}$. Assume $l = l_0$.

1. If $\mathcal{M}_s, \mathcal{M}_r \notin \prod_i \mathcal{M}_i$, after receiving all messages in $H \cdot s\langle p_1, p_2, l_0\langle v\rangle\rangle$, we have

$$\begin{array}{lcl} \Delta_s(s[p_1]^\bullet) & \curvearrowright & G_0 \upharpoonright p_1 \\ \Delta_r(s[p_2]^\bullet) & \curvearrowright & G_0 \upharpoonright p_2 \end{array}$$

2. If $\mathcal{M}_s \in \prod_i \mathcal{M}_i$, i.e. $\mathcal{M}_s$ approves receiving some messages in H, assume there is a message $s\langle p, p_1, l'_n\langle v'\rangle\rangle \in H$, then $\exists \mathcal{M}_r \in N'$ and $\exists G$ is well-formed and

$$\begin{array}{rcl} G & = & p_1 \to p_2 : \{l_k(x_k : S_k)\langle\!\langle A_k \rangle\!\rangle.G_k\}_{k\in I} \\ & = & p_1 \to p_2 : \{l_k(x_k : S_k)\langle\!\langle A_k \rangle\!\rangle.p \to p_1 : \{l'_j(x'_j : S'_j)\langle\!\langle B_j \rangle\!\rangle.G'_{kj}\}_{j\in I}\}_{k\in I} \end{array}$$

such that

$$\begin{array}{lcll} \Delta_s(s[p_1]^\bullet) & \curvearrowright & G \upharpoonright p_1 & = p_2!\{l_k(x_k : S_k)\langle\!\langle A_k \rangle\!\rangle.p?\{l'_j(x'_j : S'_j)\langle\!\langle B'_j \rangle\!\rangle.G'_{kj} \upharpoonright p_1\}_{j\in I}\}_{k\in I} \\ \Delta_r(s[p_2]^\bullet) & \curvearrowright & G \upharpoonright p_2 & = p_1?\{l_k(x_k : S_k)\langle\!\langle A'_k \rangle\!\rangle.G'_{km} \upharpoonright p_2\}_{k\in I} \end{array}$$

where $G'_{km} \upharpoonright p_2 = (p \to p_1 : \{l'_j(x'_j : S'_j)\langle\!\langle B_j \rangle\!\rangle.G'_{kj}\}_{j\in I}) \upharpoonright p_2$. After $p?$ fires for

receiving $s\langle p, p_1, l'_n\langle v'\rangle\rangle \in H$, we have

$$G \xrightarrow{s[p,p_1]?l'_n(v')}_{\mathrm{g}} G' = p_1 \to p_2 : \{l_k(x_k : S_k)\langle\!\langle A_k\rangle\!\rangle.G'_{kn}\}_{k\in I}$$

and

$$\Delta_s(s[p_1]^\bullet) \curvearrowright p_2!\{l_k(x_k : S_k)\langle\!\langle A_k\rangle\!\rangle.G'_{kn} \upharpoonright p_1\}_{k\in I}$$

since G is well-formed, G'_{kn} is well-formed. Note that G'_{kn}, n is a constant, only k is a variable, thus G'_{kn} can be viewed as G''_k. Also note that, $G'_{km} \upharpoonright p_2 = G'_{kn} \upharpoonright p_2$ because, after receiving all messages in H, the monitors in N are consistent (by Definition 7.4.12), there exists a global specification, i.e. G', such that

$$\begin{array}{rcrcl} \Delta_s(s[p_1]^\bullet) & \curvearrowright & G' \upharpoonright p_1 & = & p_2!\{l_k(x_k : S_k)\langle\!\langle A_k\rangle\!\rangle.G'_{kn} \upharpoonright p_1\}_{k\in I} \\ \Delta_r(s[p_2]^\bullet) & \curvearrowright & G' \upharpoonright p_2 & = & p_1?\{l_k(x_k : S_k)\langle\!\langle A'_k\rangle\!\rangle.G'_{kn} \upharpoonright p_2\}_{k\in I} \end{array}$$

If there are other messages in H which can be received at $\Delta_s(s[p_1]^\bullet)$, then permute every such input action, say $s[p', p_1]?l''_{n'}(v)$, until it positions after action $p_2!$ such that

$$G \curvearrowright p_1 \to p_2 : \{l_k(x_k : S_k)\langle\!\langle A_k\rangle\!\rangle.p' \to p_1 : \{l''_i(x''_j : S''_i)\langle\!\langle B'_i\rangle\!\rangle.G''_{ki}\}_{i\in I}\}_{k\in I}$$

$$\begin{array}{rcl} \Delta_s(s[p_1]^\bullet) & \curvearrowright & p_2!\{l_k(x_k : S_k)\langle\!\langle A_k\rangle\!\rangle.p'?\{l''_i(x'_j : S''_i)\langle\!\langle B''_i\rangle\!\rangle.G''_{ki} \upharpoonright p_1\}_{i\in I}\}_{k\in I} \\ \Delta_r(s[p_2]^\bullet) & \curvearrowright & p_1?\{l_k(x_k : S_k)\langle\!\langle A'_k\rangle\!\rangle.G''_{km} \upharpoonright p_2\}_{k\in I} \end{array}$$

after input $s[p', p_1]?l''_{n'}(v)$ takes place at $\Delta_s(s[p_1]^\bullet)$, we have

$$\Delta_s(s[p_1]^\bullet) \curvearrowright p_2!\{l_k(x_k : S_k)\langle\!\langle A_k\rangle\!\rangle.G''_{kn'} \upharpoonright p_1\}_{k\in I}.$$

note that, again, $G''_{km} \upharpoonright p_2 = G''_{kn'} \upharpoonright p_2$ with the same reasonings above. For every input action positioning after $p_2!$ at $\Delta_s(s[p_1]^\bullet)$, we have a general result such that there exist $\mathcal{M}_s$ and $\mathcal{M}_r$ approving action $s[p_1, p_2]!l\langle v\rangle$ to fire and to receive the generated message, and, after receiving messages in H, the resulting configurations of $\Delta_s(s[p_1]^\bullet)$ and $\Delta_r(s[p_2]^\bullet)$ should be projected from the same global specification.

3. If $\mathcal{M}_r \in \prod_i \mathcal{M}_i$, $\exists G$ is well-formed and

$$G \curvearrowright p \to p_2 : \{l'_j(x'_j : S'_j)\langle\!\langle B_j\rangle\!\rangle.p_1 \to p_2 : \{l_k(x_k : S_k)\langle\!\langle A_k\rangle\!\rangle.G'_{kj}\}_{k\in I}\}_{j\in I}$$

such that

$$\Delta_r(s[p_2]^\bullet) \curvearrowright G \upharpoonright p_2 = p?\{l'_j(x'_j : S'_j)\langle\langle B'_j\rangle\rangle.p_1?\{l_k(x_k : S_k)\langle\langle A'_k\rangle\rangle.G'_{kj} \upharpoonright p_2\}_{k\in I}\}_{j\in I}.$$

When messages in H have been received, action $s[p, p_2]?l'_n(v)$ has happened, we have

$$G \xrightarrow{s[p,p_2]?l'_n(v)} G' = p_1 \to p_2 : \{l_k(x_k : S_k)\langle\langle A_k\rangle\rangle.G'_{kn}\}_{k\in I}$$

such that

$$\Delta_r(s[p_2]^\bullet) \quad \curvearrowright \quad G' \upharpoonright p_2 = p_1?\{l_k(x_k : S_k)\langle\langle A'_k\rangle\rangle.G'_{kn} \upharpoonright p_2\}_{k\in I}$$

and since, after receiving messages in H, the monitors in N are consistent, we have

$$\Delta_s(s[p_1]^\bullet) \quad \curvearrowright \quad G' \upharpoonright p_1 = p_2!\{l_k(x_k : S_k)\langle\langle A_k\rangle\rangle.G'_{kn} \upharpoonright p_1\}_{k\in I}$$

where n is a constant.

On the above, we show that, after receiving all messages in H, the configurations at $\Delta_s(s[p_1]^\bullet)$ and $\Delta_r(s[p_2]^\bullet)$ in any case are in the form

$$\begin{aligned} \Delta_s(s[p_1]^\bullet) \quad &\curvearrowright \quad p_2!\{l_k(x_k : S_k)\langle\langle A_k\rangle\rangle.G'''_k \upharpoonright p_1\}_{k\in I} \\ \Delta_r(s[p_2]^\bullet) \quad &\curvearrowright \quad p_1?\{l_k(x_k : S_k)\langle\langle A'_k\rangle\rangle.G'''_k \upharpoonright p_2\}_{k\in I} \end{aligned} \tag{B.4}$$

which imply, whenever action $s[p_1, p_2]!l\langle v\rangle$ then action $s[p_1, p_2]?l\langle v\rangle$ happen, i.e. when messages in $H{\cdot}s\langle p_1, p_2, l\langle v\rangle\rangle$ are received, there exist G'''_k satisfying rule 4. of Definition 7.4.12. As for rule 3. of consistency of resulting monitors, since Equation B.4 is consistent, according to rule 3. of Definition 7.4.12, if there is an output in $G'''_k \upharpoonright p_2$ or in $G'''_k \upharpoonright p_1$ for $k \in I$, there exists a corresponding input in other monitors, such that when actions $s[p_1, p_2]!l\langle v\rangle$ and $s[p_1, p_2]?l\langle v\rangle$ fire, the left monitors are still consisent. Thus N' is coherent.

case (input) - if ℓ is $s[p_1, p_2]?l(v)$, since N is receivable, by Lemma 7.4.7, N' is receivable. For an input action $s[p_1, p_2]?l(v)$, only when v is a shared name, say a, will affect rules 1. and 2. Since a received name is always typed with output mode, rule 1.(a) holds. Since N is coherent, when the process who sends a has an output mode a, say $a : \mathtt{O}(G[p])$, there exists an input mode a, say $a : \mathtt{im}(G[p])$ in some other monitor; when the process who sends a has an input mode a, say $a : \mathtt{im}(G[p])$, the sender itself has a with input mode. For both cases, rule 2. holds as long as typing the received

name with output mode. For rules 3. and 4. of Definition 7.4.12, the discussions are as follows.

Suppose ℓ is $s[p_1, p_2]?l(v)$, where v is either a value or a name, and N is coherent, when $N \xrightarrow{\ell}_{\mathrm{g}} N'$:

1. If there is no pending message in N', N' is coherent because N is coherent and after N receives $s\langle p_1, p_2, l\langle v\rangle\rangle$ by ℓ, it is still coherent, which is N'.

2. If there are pending messages in N', by Lemma of receivability, N' is receivable. Let $N = \prod_{i\neq r} \mathcal{M}_i[P_i] \parallel \mathcal{M}_r[P_r] \parallel H \cdot s\langle p_1, p_2, l\langle v\rangle\rangle$, where $\mathcal{M}_r[P_r]$ will receive message $s\langle p_1, p_2, l\langle v\rangle\rangle$.

   (a) If all messages in H are received by the monitored processes in $\prod_{i\neq r} \mathcal{M}_i[P_i]$ and $\mathcal{M}_r[P_r] \xrightarrow{s[p_1,p_2]?l(v)} \mathcal{M}'_r[P'_r]$, then $N' = \prod_{i\neq r} \mathcal{M}_i[P_i] \parallel \mathcal{M}'_r[P'_r] \parallel H$. The conherence of N means that, after receiving all messages of $H\cdot s\langle p_1, p_2, l\langle v\rangle\rangle$, the network becomes $\prod_{i\neq r} \mathcal{M}'_i[P'_i] \parallel \mathcal{M}'_r[P'_r] \parallel \emptyset$, where the group of monitors is consistent; since, after N' receives all messages in H, the network also becomes $\prod_{i\neq r} \mathcal{M}'_i[P'_i] \parallel \mathcal{M}'_r[P'_r] \parallel \emptyset$, where the group of monitors is thus consistent, so that N' is coherent.

   (b) If some messages in H are received by $\prod_{i\neq r} \mathcal{M}_i[P_i]$ and some are received by $\mathcal{M}'_r[P'_r]$

      i. The configuration at $\Delta_r(s[p_2]^\bullet)$ before action $s[p_1, p_2]?l(v)$ should be in the following form:

$$\begin{array}{lll} \Delta_r(s[p_2]^\bullet) & \curvearrowright & p_1?\{l_k(x_k : S_k)\langle\langle A'_k\rangle\rangle. \\ & & p?\{l'_j(x'_j : S'_j)\langle\langle B'_j\rangle\rangle. \\ & & p'?\{l''_i(x''_i : S''_i)\langle\langle B''_i\rangle\rangle.....\}_{i\in I}\}_{j\in I}\}_{k\in I} \end{array}$$

      Assume after $\mathcal{M}_r[P_r]$ receives messages inputting to it through

$$\mathcal{M}_r[P_r] \xrightarrow{\ell} \mathcal{M}'_r[P'_r] \xrightarrow{\ell'_0 \ldots \ell'_m} \mathcal{M}''_r[P''_r],$$

      it becomes $\mathcal{M}''_r[P''_r]$. The conherence of N means that, after receiving all messages of $H \cdot s\langle p_1, p_2, l\langle v\rangle\rangle$, the network becomes $\prod_{i\neq r} \mathcal{M}'_i[P'_i] \parallel \mathcal{M}''_r[P''_r] \parallel \emptyset$, where the group of monitors is consistent. Since we have $N' = \prod_{i\neq r} \mathcal{M}_i[P_i] \parallel \mathcal{M}'_r[P'_r] \parallel H$, when N' receives all messages in H, the network also becomes $\prod_{i\neq r} \mathcal{M}'_i[P'_i] \parallel \mathcal{M}''_r[P''_r] \parallel \emptyset$, where the group of monitors is thus consistent, so that N' is coherent.

ii. before $\mathcal{M}_r[P_r] \xrightarrow{s[p_1,p_2]?l(v)} \mathcal{M}'_r[P'_r]$. Since action $s[p_1,p_2]?l(v)$ can fire immediately, the action $p_1?$ at $\Delta_r(s[p_2]^\bullet)$ can be permuted to the head:

$$\begin{array}{rcl}\Delta_r(s[p_2]^\bullet) & = & p?\{l'_j(x'_j : S'_j)\langle\!\langle B'_j\rangle\!\rangle.p_1?\{l_k(x_k : S_k)\langle\!\langle A'_k\rangle\!\rangle...\}_{k\in I}\}_{j\in I} \\ & \curvearrowright & p_1?\{l_k(x_k : S_k)\langle\!\langle A'_k\rangle\!\rangle.p?\{l'_j(x'_j : S'_j)\langle\!\langle B'_j\rangle\!\rangle...\}_{j\in I}\}_{k\in I}\end{array}$$

where $p?$ indicates receiving a message sent from p. Then the configuration goes back to the above case, so that N' is coherent.

case (acc-b/f-in) - If ℓ is $a(s[p] : G)$ or $a\langle s[p] : G\rangle$, by rule $\lfloor$MG-ACC-B/F-IN$\rfloor$, assume $N = \prod_{i\neq r} \mathcal{M}_i[P_i] \parallel \mathcal{M}_r[P_r] \parallel H\cdot\overline{a}\langle s[p] : G\rangle$ and $\mathcal{M}_r[P_r]$ will receive message $\overline{a}\langle s[p] : G\rangle$, such that

$$N \xrightarrow{\ell}_{\mathrm{g}} N' \equiv \prod_{i\neq r} \mathcal{M}_i[P_i] \parallel \mathcal{M}'_r[P'_r] \parallel H.$$

Since N is receivable, by Lemma 7.4.7, N' is receivable. The following reasonings are similar to **case (input)**.

1. If there is no pending message in N', i.e. $H = \emptyset$, N' is coherent because N is coherent and after N receives $s\langle p_1, p_2, l\langle v\rangle\rangle$ by ℓ, it is still coherent, which is N'.
2. If there are pending messages in N' and some messages in H are received by $\prod_{i\neq r} \mathcal{M}_i[P_i]$, while some are received by $\mathcal{M}_r[P_r]$, because receiving ℓ only depends on the existence of $a : \mathtt{im}(G[p]) \in \mathcal{M}_r$, this action does not affect any session input actions, i.e. it does not change other configurations at session endpoints.

Thus, after $N \xrightarrow{\ell}_{\mathrm{g}} N'$, N' is coherent.

case (join) - If ℓ is a $\mathsf{join}(s[p])$ action, since N is receivable, by Lemma 7.4.7, N' is receivable. Action $\mathsf{join}(s[p])$ does not affect any rule of the consistency of group of monitors guarding all local processes in N, thus N' is coherent.

case (tau) - if ℓ is a τ, it does not effect the receivability or any Δ_i or Γ_i in N because no monitor does checking. Thus N' is coherent.

Therefore for any action ℓ, if N is coherent and $N \xrightarrow{\ell}_{\mathrm{g}} N'$, then N' is coherent. ■

B.3 Proofs for Global Transparency

Definition B.3.1. Let N be a monitored network. Then N is *locally conformant* if, for each monitored process $\mathcal{M}_i[P_i]$ in N, we have $\mathcal{M}_i \models \mathsf{erase}(\mathcal{M}_i)[P_i]$.

Proposition B.3.2. If N is coherent and locally conformant and $N \xrightarrow{\ell}_{\mathrm{g}} N'$ then N' is also coherent and locally conformant.

Proof. By Theorem 7.4.16, N' is coherent. Assume $\prod_i \mathcal{M}_i[P_i]$ is the composition of monitored processes in N. If $N \xrightarrow{\ell}_{\mathrm{g}} N'$, there exists $\mathcal{M}_j[P_j] \in N$ and $\mathcal{M}_j[P_j] \xrightarrow{\ell}_{\mathrm{g}} \mathcal{M}'_j[P'_j]$ such that $N = N_0 \parallel \prod_{k\neq j} \mathcal{M}_k[P_k] \parallel \mathcal{M}_j[P_j] \xrightarrow{\ell}_{\mathrm{g}} N'_0 \parallel \prod_{k\neq j} \mathcal{M}_k[P_k] \parallel \mathcal{M}'_j[P'_j] = N'$. $\mathcal{M}_j[P_j] \xrightarrow{\ell}_{\mathrm{g}} \mathcal{M}'_j[P'_j]$ also infers that $\mathcal{M}_j \xrightarrow{\ell} \mathcal{M}'_j$ and $P_j \xrightarrow{\ell} P'_j$. Since N is locally conformant, $\mathcal{M}_j \models \mathsf{erase}(\mathcal{M}_j)[P_j]$. By Lemma B.1.5, as $\mathcal{M}_j \xrightarrow{\ell} \mathcal{M}'_j$ and $\mathsf{erase}(\mathcal{M}_j) \xrightarrow{\ell} \mathcal{L}$, we have $\mathcal{L} = \mathsf{erase}(\mathcal{M}'_j)$; thus we have $\mathsf{erase}(\mathcal{M}_j)[P_j] \xrightarrow{\ell} \mathsf{erase}(\mathcal{M}'_j)[P'_j]$. Together with Definition 7.4.2 of conformance, we have $\mathcal{M}'_j \models \mathsf{erase}(\mathcal{M}'_j)[P'_j]$. Note that, except $\mathcal{M}_j$, any other monitor in N does not change its configuration; the composition of monitored processes in N' is $\prod_{k\neq j} \mathcal{M}_k[P_k] \parallel \mathcal{M}'_j[P'_j]$, and every $\mathcal{M}_k[P_k] \in N', k \neq j$ satisfies $\mathcal{M}_k \models \mathsf{erase}(\mathcal{M}_k)[P_k]$. Thus N' is locally conformant. ■

Definition B.3.3 (network erasure). Assume $N = N_1 \parallel N_2$ is given. Then we have

$$\mathsf{erase}(N) = \mathsf{erase}(N_1 \parallel N_2) = \mathsf{erase}(N_1) \parallel \mathsf{erase}(N_2).$$

Theorem 7.4.17 (global transparency). Suppose N is coherent and locally conformant. Then $N \sim \mathsf{erase}(N)$.

Proof. For a network N which is coherent and locally conformant, we define relation $\mathcal{R}$ to be:

$$\mathcal{R} = \{(N, \mathsf{erase}(N)) \mid N \text{ is coherent and locally conformant}\}$$

We prove that $\mathcal{R}$ is a standard strong bisimilar relation over $\xrightarrow{\ell}_{\mathrm{g}}$. Since N is locally conformant, $\forall \mathcal{M}_i \in N, \mathcal{M}_i \models \mathsf{erase}(\mathcal{M}_i)[P_i]$, so that by Theorem 7.4.5 we have

$$\forall \mathcal{M}_i \in N, \mathcal{M}_i \models \mathsf{erase}(\mathcal{M}_i)[P_i] \sim \mathcal{M}_i[P_i].$$

1. As $N \xrightarrow{\ell}_{\mathrm{g}} N'$, it implies there is $\mathcal{M}_j[P_j] \in N$ and $\mathcal{M}_j[P_j] \xrightarrow{\ell} \mathcal{M}'_j[P'_j]$ such that $N = N_0 \parallel \mathcal{M}_j[P_j] \xrightarrow{\ell}_{\mathrm{g}} N'_0 \parallel \mathcal{M}'_j[P'_j] = N'$. $\mathcal{M}_j[P_j] \xrightarrow{\ell} \mathcal{M}'_j[P'_j]$ also infers $\mathcal{M}_j \xrightarrow{\ell} \mathcal{M}'_j$ and $P_j \xrightarrow{\ell} P'_j$. Note that N_0 is changed to N'_0 because N_0 contains the global queue, the message corresponding to ℓ may enter or leave the global queue. All monitored processes in N_0 are not affected. By Definition 7.4.4, when ℓ is an input action, $\mathcal{M}_j[P_j] \xrightarrow{\ell} \mathcal{M}'_j[P'_j]$ and $\mathcal{M}_j \xrightarrow{\ell} \mathcal{M}'_j$ imply $\mathsf{erase}(\mathcal{M}_j)[P_j] \xrightarrow{\ell} \mathsf{erase}(\mathcal{M}'_j)[P'_j]$, so that, based on Definition B.3.3, $\mathsf{erase}(N) \equiv \mathsf{erase}(N_0) \parallel \mathsf{erase}(\mathcal{M}_j)[P_j] \xrightarrow{\ell}_{\mathrm{g}}$

$\mathsf{erase}(N_0') \parallel \mathsf{erase}(\mathcal{M}_j')[P'] \equiv \mathsf{erase}(N_0' \parallel \mathcal{M}_j'[P']) = \mathsf{erase}(N')$; similarly, when ℓ is an output action, $\mathcal{M}_j[P_j] \xrightarrow{\ell} \mathcal{M}_j'[P_j']$ implies $\mathsf{erase}(\mathcal{M}_j)[P_j] \xrightarrow{\ell} \mathcal{L}$, $\mathcal{M}_j \xrightarrow{\ell} \mathcal{M}_j'$ and $P_j \xrightarrow{\ell} P_j'$; by Lemma B.1.5, $\mathsf{erase}(\mathcal{M}_j) \xrightarrow{\ell} \mathsf{erase}(\mathcal{M}_j')$, so that $\mathcal{L} = \mathsf{erase}(\mathcal{M}_j')[P_j']$. Again, we have $\mathsf{erase}(N) \xrightarrow{\ell}_{\mathrm{g}} \mathsf{erase}(N')$. By proposition B.3.2, N' is coherent and locally conformant, thus we have $(N', \mathsf{erase}(N')) \in \mathcal{R}$.

2. Assume $N = N_0 \parallel \prod_k \mathcal{M}_k[P_k]$, $\mathsf{erase}(N) \xrightarrow{\ell}_{\mathrm{g}} \mathsf{N}'$ implies there is $\mathsf{erase}(\mathcal{M}_j)[P_j] \in \mathsf{erase}(N)$ and $\mathsf{erase}(\mathcal{M}_j)[P_j] \xrightarrow{\ell} \mathcal{L}'$ such that

$$\begin{array}{rcl} \mathsf{erase}(N) & = & \mathsf{erase}(N_0) \parallel \prod_{k\neq j} \mathsf{erase}(\mathcal{M}_k)[P_k] \parallel \mathsf{erase}(\mathcal{M}_j)[P_j] \xrightarrow{\ell}_{\mathrm{g}} \\ & & \mathsf{erase}(N_0') \parallel \prod_{k\neq j} \mathsf{erase}(\mathcal{M}_k)[P_k] \parallel \mathcal{L}' = \mathsf{N}' \end{array}$$

Note that, since $\mathsf{erase}(N_0)$ contains the global queue, it changes to $\mathsf{erase}(N_0')$ as action ℓ happens. All located processes in $\mathsf{erase}(N_0)$ are not affected. By Definition 7.4.4, when ℓ is an input action and $\mathcal{M}_j \models \mathsf{erase}(\mathcal{M}_j)[P_j]$, $\mathsf{erase}(\mathcal{M}_j)[P_j] \xrightarrow{\ell}$ infers $\mathcal{M}_j \xrightarrow{\ell}$ and $\mathsf{erase}(M_j) \xrightarrow{\ell}$, which means $\mathcal{M}_j$ and $\mathsf{erase}(\mathcal{M}_j)$ approve action ℓ. Without loss of generality, let $\mathcal{M}_j \xrightarrow{\ell} \mathcal{M}_j'$. By Lemmas B.1.3 and B.1.5, $\mathcal{M}_j \xrightarrow{\ell} \mathcal{M}_j'$ implies that, if $\mathsf{erase}(\mathcal{M}_j) \xrightarrow{\ell}$, then $\mathsf{erase}(\mathcal{M}_j) \xrightarrow{\ell} \mathsf{erase}(\mathcal{M}_j')$. Together with $\mathsf{erase}(\mathcal{M}_j)[P_j] \xrightarrow{\ell} \mathcal{L}'$, we have $\mathcal{L}' = \mathsf{erase}(\mathcal{M}_j')[P']$ for some P'. Since $\mathcal{M}_j \models \mathsf{erase}(\mathcal{M}_j)[P] \sim \mathcal{M}_j[P_j]$, $\mathsf{erase}(\mathcal{M}_j)[P_j] \xrightarrow{\ell} \mathsf{erase}(\mathcal{M}_j')[P_j']$ and $\mathcal{M}_j \xrightarrow{\ell} \mathcal{M}_j'$ imply $\mathcal{M}_j[P_j] \xrightarrow{\ell} \mathcal{M}_j'[P_j']$, so that $N = N_0 \parallel \prod_{k\neq j} \mathcal{M}_k[P_k] \parallel \mathcal{M}_j[P_j] \xrightarrow{\ell}_{\mathrm{g}} N_0' \parallel \prod_{k\neq j} \mathcal{M}_k[P_k] \parallel \mathcal{M}_j'[P_j'] = N'$; similarly, when ℓ is an output action and $\mathcal{M}_j \models \mathsf{erase}(\mathcal{M}_j)[P_j]$, $\mathsf{erase}(\mathcal{M}_j)[P_j] \xrightarrow{\ell}$ implies $\mathcal{M}_j \xrightarrow{\ell}$ and $\mathcal{M}_j[P_j] \xrightarrow{\ell}$. Without loss of generality, let $\mathcal{M}_j \xrightarrow{\ell} \mathcal{M}_j'$. By Lemmas B.1.3 and B.1.5, we have $\mathcal{M}_j[P_j] \xrightarrow{\ell} \mathcal{M}_j'[P_j']$ for some P_j', so that we have $N = N_0 \parallel \prod_{k\neq j} \mathcal{M}_k[P_k] \parallel \mathcal{M}_j[P_j] \xrightarrow{\ell}_{\mathrm{g}} N_0' \parallel \prod_{k\neq j} \mathcal{M}_k[P_k] \parallel \mathcal{M}_j'[P_j'] = N'$. Since $\mathcal{L}' = \mathsf{erase}(\mathcal{M}_j')[P_j']$, by Definition B.3.3, $\mathsf{erase}(N') = \mathsf{erase}(N_0' \parallel \prod_{k\neq j} \mathcal{M}_k[P_k] \parallel \mathcal{M}_j'[P_j]) \equiv \mathsf{erase}(N_0') \parallel \prod_{k\neq j} \mathsf{erase}(\mathcal{M}_k)[P_k] \parallel \mathsf{erase}(\mathcal{M}_j')[P_j] = \mathsf{erase}(N_0') \parallel \prod_{k\neq j} \mathsf{erase}(\mathcal{M}_k)[P_k] \parallel \mathcal{L}' = \mathsf{N}'$. By proposition B.3.2, N' is coherent and locally conformant, thus we have $(N', \mathsf{N}') = (N', \mathsf{erase}(N')) \in \mathcal{R}$. ■

B.4 Proofs for Session Fidelity

The theorem of session fidelity states that, whenever a network conforms to specifications in monitors, i.e., its all local processes (partial network) which are in $\mathsf{P}(\mathsf{N})$ conform to specifcations contained in $\mathbb{M}$, all of its derivatives conform to the specifications of

$\mathbb{M}$. In the follows, we firstly formally define *receivability, consistency and conformance* based on LTS of configurations. Recall that a *configuration* is the pair of $\mathbb{M}$ and H:

Definition 7.4.20 (configuration). A configuration is denoted by $\Phi = \mathbb{M}; H$, in which the group of monitors correspond to H. In other words, all messages corresponding to the actions guarded by $\mathbb{M}$ are in H.

Definition B.4.1 (receivable configuration). Define $\mathbb{M}; H$ is *receivable* by the following induction.

1. If H is empty then $\mathbb{M}; H$ is receivable.
2. If $H \equiv m \cdot H'$, then $\mathbb{M}; H$ is receivable when we have $\mathbb{M}; m \cdot H' \xrightarrow{\ell}_{\mathrm{g}} \mathbb{M}'; H'$, where ℓ corresponding to m, and $\mathbb{M}'; H'$ is receivable.

A configuration $\mathbb{M}; H$ is *configurationally consistent* if all of its multi-step global input transition derivatives are receivable and the resulting $\mathbb{M}$ is consistent. The consistency of a group of monitors is defined in Definition 7.4.12.

Lemma B.4.2. Assume $\mathsf{N} \equiv \mathsf{N}_s \parallel H$ and $\models \mathsf{N}_s : \mathbb{M}$. If $\mathsf{N} \xrightarrow{\ell}_{\mathrm{g}} \mathsf{N}' \equiv \mathsf{N}_s{}' \parallel H'$ and $\mathbb{M} \xrightarrow{\ell} \mathbb{M}'$, then $\models \mathsf{N}_s{}' : \mathbb{M}'$.

Proof. Directly from Definition 7.4.25.■

Session fidelity says: assume a network $\mathsf{N} \equiv \mathsf{N}_s \parallel \emptyset$ is given, and suppose that N_s satisfies $\mathbb{M}$. If $\mathbb{M}$ is consistent, then we say N *conforms to* $\mathbb{M}$. If this holds, then, with the messages which N exchanges follow the specification containing in $\mathbb{M}$, *the dynamics of the network* witnesses the validity of $\mathbb{M}$.

Theorem 7.4.28 (session fidelity). Assume configuration $\mathbb{M}; H$ is configurationally consistent, and network $\mathsf{N} \equiv \mathsf{N}_s \parallel H$ conforms to configuration $\mathbb{M}; H$. We say N satisfies session fidelity, if for any ℓ, we have $\mathsf{N} \xrightarrow{\ell}_{\mathrm{g}} \mathsf{N}'$ such that $\mathbb{M}; H \xrightarrow{\ell}_{\mathrm{g}} \mathbb{M}'; H'$, it holds that $\mathbb{M}'; H'$ is configurationally consistent and that N' conforms to $\mathbb{M}'; H'$.

Proof. Assume $\mathsf{N} \equiv \mathsf{N}_s \parallel H$ and N conforms to $\mathbb{M}; H$, which is configurationally consistent. Prove the statement by the inspection of each case. (In most cases, the proofs are similar to those for Theorem 7.4.16, proved in Appendix B.2).

(Sel) Let $\ell = s[p_1, p_2]!l_j\langle v\rangle$, $\mathsf{N}_s \parallel H \equiv \mathsf{N} \xrightarrow{\ell}_{\mathrm{g}} \mathsf{N}' \equiv \mathsf{N}_s{}' \parallel H \cdot m$ and $\mathbb{M}; H \xrightarrow{\ell}_{\mathrm{g}} \mathbb{M}'; H \cdot m$, where $m = s\langle p_1, p_2, l_j\langle v\rangle\rangle$. Since $\mathbb{M}$ allows ℓ and $\mathbb{M}$ is consistent, $\exists \mathcal{M}_s, \mathcal{M}_r \in \mathbb{M}$,

$\exists G$ is well-formed such that

$$G = p_1 \to p_2\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G_i\}_{i\in I},$$

and

$$\begin{aligned}\Delta_s(s[p_1]^\bullet) &\curvearrowright G \upharpoonright p_1 = p_2!\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G_i \upharpoonright p_1\}_{i\in I},\\ \Delta_r(s[p_2]^\bullet) &\curvearrowright G \upharpoonright p_2 = p_1?\{l_i(x_i : S_i)\langle\langle A'_i\rangle\rangle.G_i \upharpoonright p_2\}_{i\in I}.\end{aligned} \tag{B.5}$$

where $l_j \in \{l_i\}_{i\in I}$, I is the set indexing labels. Note that, the configuration of $\Delta_r(s[p_2]^\bullet)$ is not changed by action ℓ, thus $\Delta_r(s[p_2]^\bullet)$ remains the same in $\mathbb{M}'$.

Consider (case 1: H is empty). Since $\mathbb{M}'$ has $\Delta_r(s[p_2]^\bullet) \curvearrowright p_1?\{l_i(x_i : S_i)\langle\langle A'_i\rangle\rangle.G_i \upharpoonright p_2\}_{i\in I}$, m is receivable to $\mathbb{M}'$, and thus m is receivable to $\mathsf{N}_s{}'$ due to $\models \mathsf{N}_s{}' : \mathbb{M}'$, according to Lemma B.4.2. After receiving m, say $\mathbb{M}' \xrightarrow{s[p_1,p_2]?l_j(v)} \mathbb{M}''$, according to Equation B.5, $\mathbb{M}''$ has $\Delta_s(s[p_1]^\bullet) \curvearrowright G_j \upharpoonright p_1$ and $\Delta_r(s[p_2]^\bullet) \curvearrowright G_j \upharpoonright p_2$, and action $s[p_1, p_2]?l_j(v)$ does not affect any other configuration in $\mathbb{M}'$, $\mathbb{M}''$ is consistent by Definition 7.4.12. Therefore, N' conforms to $\mathbb{M}'; m$.

Consider (case 2: H is not empty). Since $\mathsf{N} \equiv \mathsf{N}_s \parallel H$ conforms to $\mathbb{M}; H$, all messages in H are receivable to N_s. With the same reasonings of the proof for **output** in Theorem 7.4.16 for global safety, whenever $\mathbb{M}; H \xrightarrow{s[p_1,p_2]!l_j\langle v\rangle}_{\mathrm{g}} \mathbb{M}'; H \cdot s\langle p_1, p_2, l_j\langle v\rangle\rangle$, $\mathbb{M}'$ is consistent and $H \cdot s\langle p_1, p_2, l_j\langle v\rangle\rangle$ are receivable to $\mathbb{M}'$ thus to N'. Therefore, N' conforms to $\mathbb{M}'; H \cdot s\langle p_1, p_2, l_j\langle v\rangle\rangle$.

(SelN) When $\ell = s[p_1, p_2]!l_j\langle a\rangle$, where a is a name, by the LTS of monitors defined in Figure 7.4, this case is as same as the case of **(Sel)**.

(Bra) Let $\ell = s[p_1, p_2]?l_j(v)$. Consider (case 1: H is empty). Since $\mathbb{M}; \emptyset \overset{\ell}{\not\to}_{\mathrm{g}}$, this case never happens.

Consider (case 2: H is not empty). When H is not empty. $\mathsf{N} \xrightarrow{\ell}_{\mathrm{g}} \mathsf{N}'$ and

$$\mathbb{M}; H \xrightarrow{\ell}_{\mathrm{g}} \mathbb{M}'; H/m,$$

where H/m means taking off message m from H, $m = s\langle p_1, p_2, l_j\langle v\rangle\rangle$. By Definition 7.4.24, with the same reasonings of the proof for **input** in Theorem 7.4.16 for global safety, since $\mathbb{M}$ is consistent after receiving all messages H, $\mathbb{M}'$ should

be consistent after receiving messages H/m. Thus we have $\mathbb{M}'; H/m$ is configurationally consistent, to which we can check N' conforms.

(BraN) When $\ell = s[p_1, p_2]?l_j(a)$, where a is a name, by the LTS of monitors defined in Figure 7.4, this case is similar to the case of (Bra) except that the received a is typed with $\mathtt{0}$. Since this typing does not affect any consistency rule, it is proved as case (**Bra**).

(Req-b) Let $\ell = \overline{a}(s[p_k] : G)$. $\mathsf{N} \xrightarrow{\ell}_{\mathrm{g}} \mathsf{N}' \equiv \mathsf{N}_s{}' \parallel H \cdot m$ as well as $\mathbb{M}; H \xrightarrow{\ell}_{\mathrm{g}} \mathbb{M}'; H \cdot m$, where $m = \overline{a}\langle s[p_k] : G\rangle$. This action combines two parts: creating a fresh session and requesting through sending an invitation. As for the part of creating a session, the reasonings are as same as those of the proof for **bound request** in Theorem 7.4.16 for global safety.

As for the part of requesting an invitation to endpoint a and other parts of proving that the resulting dynamic network conforms to the resulting configuration, the reasonings are as same as that for **(Req-f)** below.

(Req-f) Let $\ell = \overline{a}\langle s[p] : G\rangle$. $\mathsf{N} \xrightarrow{\ell}_{\mathrm{g}} \mathsf{N}' \equiv \mathsf{N}_s{}' \parallel H \cdot m$ as well as $\mathbb{M}; H \xrightarrow{\ell}_{\mathrm{g}} \mathbb{M}'; H \cdot m$, where $m = \overline{a}\langle s[p] : G\rangle$. Since $\mathbb{M}$ allows ℓ and $\mathbb{M}$ is consistent, $\exists \mathcal{M}_j \in \mathbb{M}$ such that $a : \mathtt{0}(G[p]) \in \Gamma_j$ or $a : \mathtt{IO}(G[p]) \in \Gamma_j$, and $\exists \mathcal{M}_r \in \mathbb{M}$ such that $a : \mathtt{im}(G[p]) \in \Gamma_r$. Thus, $\mathcal{M}_r$ is able to receive the message $m = \overline{a}\langle s[p] : G\rangle$ created by ℓ, which implies that m is receivable to $\mathbb{M}'$ thus receivable to N' because of $\models \mathsf{N}_s{}' : \mathbb{M}'$.

After $\mathbb{M} \xrightarrow{\ell} \mathbb{M}'$ (generating m by requesting at $\mathcal{M}_j$) and $\mathbb{M}' \xrightarrow{\ell'} \mathbb{M}''$ (receiving m by accepting at $\mathcal{M}_r$), where $\ell' = a(s[p] : G)$ or $\ell' = a\langle s[p] : G\rangle$, according to rules [Req-b] and [Acc-f/b] in Figure 7.4, $a : \mathtt{0}(G[p]) \in \Gamma_j$ or $a : \mathtt{IO}(G[p]) \in \Gamma_j$, and $a : \mathtt{im}(G[p]) \in \Gamma_r$ all remain in $\mathbb{M}'$ and $\mathbb{M}''$. Moreover, since the capability $s[p] : G \upharpoonright p$ is exchanged from $\mathcal{M}_j$ to $\mathcal{M}_r$ by removing it from $\mathcal{M}_j$ and adding it to $\mathcal{M}_r$, all rules about the capability of $s[p]$ defined in Definition 7.4.12 still hold for $\mathbb{M}'$ and $\mathbb{M}''$. Thus $\mathbb{M}'$ and $\mathbb{M}''$ are consistent. By Lemma B.4.2, we have $\models \mathsf{N}_s{}' : \mathbb{M}'$. Since all messages in H are receivable to $\mathbb{M}$, and ℓ does not affect any other configurations except those explained above, all messages in H are receivable to $\mathsf{N}_s{}'$. Because m is receivable to $\mathsf{N}_s{}'$ as we proved above, $H \cdot m$ is receivable to $\mathsf{N}_s{}'$. In conclusion, N' conforms to $\mathbb{M}'; H \cdot m$, where $m = \overline{a}\langle s[p] : G\rangle$.

(Acc-b/f) Let $\ell = a(s[p] : G)$ or $\ell = a\langle s[p] : G\rangle$.

Consider (case 1: H is empty). Since $\mathbb{M}; \emptyset \overset{\ell}{\not\to}_{\mathrm{g}}$, this case never happens.

Consider (case 2: H is not empty). If $\mathsf{N}_s \cdot H \equiv \mathsf{N} \xrightarrow{\ell}_{\mathrm{g}} \mathsf{N}' \equiv \mathsf{N}_s{}' \parallel H/m$ and $\mathbb{M}; H \xrightarrow{\ell}_{\mathrm{g}} \mathbb{M}'; H/m$ where $m = \overline{a}\langle s[p] : G\rangle$, they imply $\exists \mathcal{M}_r \in \mathbb{M}$ and $\mathcal{M}_r \xrightarrow{\ell} \mathcal{M}'_r$, such that $a : \mathtt{im}(G[p]) \in \Gamma_r$. Since all messages in H are receivable to $\mathbb{M}$ and ℓ only receives m from H and added capability of $s[p]$ into $\mathcal{M}'_r \in \mathbb{M}'$ without affecting the receivability of other messages in H, the left messages in H/m are receivable to $\mathsf{N}_s{}'$. By Lemma B.4.2, we have $\models \mathsf{N}_s{}' : \mathbb{M}'$. In conclusion N' conforms to $\mathbb{M}'; H/m$, where $m = \overline{a}\langle s[p] : G\rangle$.

The proof for other cases are trivial. ■

B.5 Proofs for $\mathfrak{E}$ Coherence

Before proving Poposition 7.5.7 for the property of $\mathfrak{E}$ coherence, the following definitions and lemmas are introduced. We define $G' \subseteq G$ w.r.t a particular session similarly as the one for local specifications defined in Definition 7.4.9.

Definition B.5.1 ($G' \subseteq G$ w.r.t a session). We say $G' \subseteq G$ w.r.t s if, for some $\mathfrak{E}$ containing G, we have either (I) $s : G \xrightarrow{\ell_1 \ldots \ell_n} s : G'$ for some sequence of actions $\ell_1 \ldots \ell_n$ or (II) $G \curvearrowright G'$.

Note that, when G, G' are clearly for the same session, we simply write $G' \subseteq G$.

Define the result after kth valid unit permutation of a global specification:

Definition B.5.2. Assume $p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G_i\}_{i\in I} \subseteq G$. Define $p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G_i^k\}_{i\in I}$ is the result after kth valid unit permutation of $p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G_i\}_{i\in I}$ with its k previous interactions.

Similarly, define the result after kth valid unit permutation of a local specification:

Definition B.5.3. Assume $q!\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.T_i\}_{i\in I} \subseteq T$. Define $q!\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.T_i^k\}_{i\in I}$ is the result after kth valid unit permutation of $q!\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.T_i\}_{i\in I}$ with its k previous interactions. Similarly, for the case $q?\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.T_i\}_{i\in I} \subseteq T$, define $q?\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.T_i^k\}_{i\in I}$ is the result after kth valid unit permutation of $q?\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.T_i\}_{i\in I}$ with its k previous interactions.

Example B.5.4. Assume an interaction

$$p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G_i\}_{i\in I} \subseteq G,$$

is the kth interaction in G, and let G be in the following shape:

$$G = p_k \to q_k : \{..... .p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G_i\}_{i\in I}\}...\}...\}.$$

where the interaction with prefix $p_k \to q_k$ is the first interaction in G.

$$G \curvearrowright p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G'_i\}_{i\in I}$$

according to Definition 6.6.5, it means that we should be able to permute interaction with prefix $p \to q$ to the top of G after k valid unit permutations.

Assume the last interaction sequenced before $p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G_i\}_{i\in I}$ is interaction with prefix $p_1 \to q_1$ where $q_1 \neq p$ or $(p_1 \neq p) \cap (q_1 \neq q)$:

$$\begin{array}{l} p_1 \to q_1 : \{l'_{j'}(x'_{j'} : S'_{j'})\langle\langle A'_{j'}\rangle\rangle.G'_{j'}\}_{j'\in J} = \\ p_1 \to q_1 : \{l'_{j'}(x'_{j'} : S'_{j'})\langle\langle A'_{j'}\rangle\rangle.p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G_{j'i}\}_{i\in I}\}_{j'\in J} \end{array}$$

where $G'_{j'} = p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G_{j'i}\}_{i\in I}$ and $G_{j'i} = G_{ij'}$. By Definition 6.6.5,

$$\begin{array}{l} p_1 \to q_1 : \{l'_{j'}(x'_{j'} : S'_{j'})\langle\langle A'_{j'}\rangle\rangle.p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G_{j'i}\}_{i\in I}\}_{j'\in J} \curvearrowright \\ p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.p_1 \to q_1 : \{l'_{j'}(x'_{j'} : S'_{j'})\langle\langle A'_{j'}\rangle\rangle.G_{ij'}\}_{j'\in J}\}_{i\in I} = \\ p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G^1_i\}_{i\in I}, \end{array}$$

where for all $i \in I$, $G^1_i = p_1 \to q_1 : \{l'_{j'}(x'_{j'} : S'_{j'})\langle\langle A'_{j'}\rangle\rangle.G_{ij'}\}_{j'\in I}$. Note that $p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G^1_i\}_{i\in I}$ results the 1st permutation of $p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G_i\}_{i\in I}$ with its 1st previous interactions.

Now we state that $p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G^{k-1}_i\}_{i\in I}$ is the result after $k-1$ valid permutations from $p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G_i\}_{i\in I}$. Then we can permute $p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G_i\}_{i\in I}$ to the top after 1 unit valid permutations:

$$\begin{array}{l} p_k \to q_k : \{l''_j(x''_j : S''_j)\langle\langle A''_j\rangle\rangle.p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G^{k-1}_{ij}\}_{i\in I}\}_{j\in J} \curvearrowright \\ p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.p_k \to q_k : \{l''_j(x''_j : S''_j)\langle\langle A''_j\rangle\rangle.G^{k-1}_{ji}\}_{j\in J}\}_{i\in I} = \\ p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G^k_i\}_{i\in I} \end{array}$$

where $G^{k-1}_{ji} = G^{k-1}_{ij} = G^{k-1}_i$.

Lemma B.5.5. Assume $p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G_i\}_{i\in I} \subseteq G$. If $G \curvearrowright p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G'_i\}_{i\in I}$, then there does not exist an interaction with prefix $q' \to p$ or $p \to q$ suppressing $p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G_i\}_{i\in I} \subseteq G$.

Proof. Assume an interaction

$$p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G_i\}_{i\in I} \subseteq G,$$

is the nth interaction in G, and let G be in the following form:

$$G = p_n \to q_n : \{..... .p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G_i\}_{i\in I}\}...\}...\}.$$

where the interaction with prefix $p_n \to q_n$ is the first interaction in G.

If there is an interaction with prefix $p' \to q'$, where $q' = p$ or $(p' = p) \cap (q' = q)$, sequenced before $p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G^k_i\}_{i\in I}$, which is the result after kth permutation of $p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G_i\}_{i\in I}$ (according to Definition B.5.2), then by Definition 6.6.5,

$$\begin{array}{l} p' \to q' : \{l'_{j'}(x'_{j'} : S'_{j'})\langle\langle A'_{j'}\rangle\rangle.G'_{j'}\}_{j'\in J} = \\ p' \to q' : \{l'_{j'}(x'_{j'} : S'_{j'})\langle\langle A'_{j'}\rangle\rangle.p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G^k_{j'i}\}_{i\in I}\}_{j'\in J} \\ \not\curvearrowright p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.p' \to q' : \{l'_{j'}(x'_{j'} : S'_{j'})\langle\langle A'_{j'}\rangle\rangle.G^k_{ij'}\}_{j'\in J}\}_{i\in I} \end{array}$$

This is a contradiction.■

Lemma B.5.6. Assume there exists $T_0 \subseteq T$.

1. Assume $T_0 = q!\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.T_i\}_{i\in I} \subseteq T$, if $T \curvearrowright q!\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.T'_i\}_{i\in I}$, then there does not exist an action with prefix $q'?$ or $q!$ suppressing $q!\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.T_i\}_{i\in I}$ in T.
2. Similarly, assume $T_0 = p?\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.T_i\}_{i\in I} \subseteq T$, if $T \curvearrowright p?\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.T'_i\}_{i\in I}$, then there does not exist an action with prefix $p?$ or p suppressing $p!\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.T_i\}_{i\in I}$ in T.

Proof. The proof is similar to the proof of Lemma B.5.5. Assume action

$$T_0 = q!\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.T_i\}_{i\in I} \subseteq T,$$

is the nth action in T and T is in the following shape

$$T = ...q!\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.T_i\}_{i\in I}\}...\}...\}.$$

If there is some action with prefix q_1 such that $q_1 = q'?$ or $q_1 = q!$ sequenced before action $q!\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.T_i^k\}_{i\in I}$, the result after kth valid unit permutation of $q!\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.T_i\}_{i\in I}$ (according to Definition B.5.3), by Definition 6.6.8,

$$q_1!\{l'_{j'}(x'_{j'} : S'_{j'})\langle\langle A'_{j'}\rangle\rangle.T'_{j'}\}_{j'\in I} =$$
$$q_1!\{l'_{j'}(x'_{j'} : S'_{j'})\langle\langle A'_{j'}\rangle\rangle.q!\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.T^k_{j'i}\}_{i\in I}\}_{j'\in I}$$

where $T'_{j'} = q!\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.T^k_{j'i}\}_{i\in I}$ and $T^k_{j'i} = T^k_{ij'} = T^k_i$. This is a contradiction.

Similarly, we can prove that for the input case, it is a contradiction too.■

Lemma B.5.7. Given $\mathfrak{E}$ is coherent, $s : G, s : \widetilde{mv} \in \mathfrak{E}, p, q \in role(G)$ and there is no $\langle p, q, l''\langle v''\rangle\rangle$ for some l'' and v'' in $\widetilde{mv}$. If $G \upharpoonright p - \widetilde{mv} \upharpoonright p \curvearrowright q!\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.T_i\}_{i\in I}$, then

$$G \curvearrowright p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G_i\}_{i\in I}, \forall\ i \in I, G_i \upharpoonright p = T_i.$$

Proof. Note that $s : \widetilde{mv}$ means that there is only $\widetilde{mv}$ belonging to session s in the global queue. We prove it by contradiction. Assume

$$G \not\curvearrowright p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G_i\}_{i\in I}, \forall\ i \in I, G_i \upharpoonright p = T_i.$$

Then Lemma B.5.5 indicates

$$\exists\ p_0 \to q_0 : \{l'_j(x'_j : S'_j)\langle\langle A'_j\rangle\rangle.G'_j\}_{j\in J} \subseteq G$$

where $q_0 = p$ or $(p_0 = p) \cap (q_0 = q)$ such that interaction $p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G_i\}_{i\in I}$ is suppressed; then Definition 6.5.1 implies that $G \upharpoonright p$ should be

$$G \upharpoonright p = ...p_0?\{l'_j(x'_j : S'_j)\langle\langle A''_j\rangle\rangle.T'_j\}_{j\in J}\}...\}...\}$$

or

$$G \upharpoonright p = ...q!\{l'_j(x'_j : S'_j)\langle\langle A''_j\rangle\rangle.T'_j\}_{j\in J}\}...\}...\}$$

if it is the kth action, where $\forall\ j \in J$, $A''_j\{e/x'_j\} \downarrow \mathsf{true}$ iff $A'_j\{e/x'_j\} \downarrow \mathsf{true}$ for some e.

For the case

$$G \upharpoonright p = ...p_0?\{l'_j(x'_j : S'_j)\langle\langle A''_j\rangle\rangle.T'_j\}_{j\in J}\}...\}...\}$$

when we use $G \upharpoonright p - \widetilde{mv} \upharpoonright p$ to remove those *happened* output actions in $G \upharpoonright p$, we can never get $G \upharpoonright p - \widetilde{mv} \upharpoonright p \curvearrowright q!\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.T_i\}_{i\in I}$,

(I) By Definition 7.5.2, $\widetilde{mv} \upharpoonright p$ only indicates messages sent from p, so $\widetilde{mv} \upharpoonright p$ cannot

remove an input action in $G \upharpoonright p$;

(II) Moreover, in $\widetilde{mv}$, there is no message of output actions which are sequenced behind action $p_0?$ because, by Definition 6.6.8, such an output only happens after action $p_0?$ fires.

(III) Together with (I), (II), we know that after doing $G \upharpoonright p - \widetilde{mv} \upharpoonright p$, the action $p_0?$ still suppresses the action $q!\{l_i(x_i : S_i)\}$ for $i \in I$. This a contradiction.

For the case

$$G \upharpoonright p = ...q!\{l'_j(x'_j : S'_j)\langle\langle A''_j\rangle\rangle.T'_j\}_{j\in J}\}...\}...\},$$

the reasoning is below:

(i) Since there is no $\langle p, q, l''\langle v''\rangle\rangle$ for some l'' and v'' in $\widetilde{mv}$, so that taking off $\widetilde{mv} \upharpoonright p$ cannot remove action $q!\{l'_j(x'_j : S'_j)\}$ for $j \in J$.

(ii) Based on (i), we know that after doing $G \upharpoonright p - \widetilde{mv} \upharpoonright p$, the action $q!\{l'_j(x'_j : S'_j)\}$ for $j \in J$ still suppresses the action $q!\{l_i(x_i : S_i)\}$ for $i \in I$. This a contradiction.

Thus

$$G \curvearrowright p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G_i\}_{i\in I}, \forall\ i \in I, G_i \upharpoonright p = T_i$$

if

$$G \upharpoonright p - \widetilde{mv} \upharpoonright p = q!\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.T_i\}_{i\in I}. \blacksquare$$

Poposition 7.5.7 If $\mathfrak{E}$ is coherent and $\mathfrak{E} \xrightarrow{\ell}_{\mathrm{g}} \mathfrak{E}'$, then $\mathfrak{E}'$ is coherent.

Proof. We illustrate the non-trivial cases of ℓ. Note that, since the first rule of $\mathfrak{E}$ coherence: $\forall a,\ a : \mathtt{om}(G[p]) \in \Gamma$ implies that $\exists \Gamma', a : \mathtt{im}(G[p]) \in \Gamma'$, is trivially true in the following cases, thus we ignore the proof of the first rule.

Case [$\mathfrak{E}$-SEL/SELN] Assume $\ell = s[p, q]!l_j\langle v\rangle$, where v is either a value of a name. Since $\mathfrak{E}$ is coherent, and $\mathfrak{E} \xrightarrow{\ell} \mathfrak{E}'$, by rule [$\mathfrak{E}$-SEL], let $\mathfrak{E} = \mathfrak{E}_0, s[p]^\bullet : T_0, s : \widetilde{mv}$, where $T_0 \curvearrowright q!\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.T_i\}_{i\in I}$, and thus $\mathfrak{E}' = \mathfrak{E}_0, s[p]^\bullet : T'_0, s : \widetilde{mv}\cdot\langle p, q, l_j\langle v\rangle\rangle$, where $T'_0 \curvearrowright T_j\{v/x_j\}$. Check if $\mathfrak{E}'$ satisfies the three other rules of $\mathfrak{E}$-coherence (Definition 7.5.6):

1. Since $\mathfrak{E}$ is coherent, $s[p]^\bullet \in \Delta$ implies $s : G \in \Theta$ and $p \in \mathit{role}(G)$. Note that, by Definition 7.5.5, whenever $s : \mathsf{end}$ (i.e. the session is finished), everything related to session s will be deleted immediately from $\mathfrak{E}$. If $s : G \notin \mathfrak{E}'$, i.e. $s : \mathsf{end}$, since the configuration of $s : G$ does not change by an output action, thus it implies

$s : G \notin \mathfrak{E}$. Since $\mathfrak{E}$ is coherent, it implies $s[p]^\bullet \notin \mathfrak{E}$, which is a contradiction. Thus $\mathfrak{E}'$ satisfies the 2nd rule of $\mathfrak{E}$-coherence.

2. Since $\mathfrak{E}$ is coherent, in $\mathfrak{E}$, $\exists\, T$ such that

$$G \upharpoonright p - \widetilde{mv} \upharpoonright p = q!\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.T_i\}_{i\in I} = T,$$

where $\Delta(s[p]^\bullet) \curvearrowright T$.

By rule [$\mathfrak{E}$-SEL], in $\mathfrak{E}'$, $\Delta(s[p]^\bullet) = T'_0 = T_j\{v/x_j\}$. Check if it satisfies $G \upharpoonright p - \widetilde{mv} \cdot \langle p, q, l_j\langle v\rangle\rangle \upharpoonright p = T_j\{v/x_j\}$.

(a) If $\widetilde{mv} \upharpoonright p = \varepsilon$, then $G \upharpoonright p - \widetilde{mv} \upharpoonright p = G \upharpoonright p = T$. By Definition 7.5.2, in $\mathfrak{E}'$ we have

$$\begin{array}{rcl} G \upharpoonright p - \widetilde{mv'} \upharpoonright p & = & G \upharpoonright p - \widetilde{mv} \upharpoonright p \cdot \langle p, q, l_j\langle v\rangle\rangle \upharpoonright p \\ & = & G \upharpoonright p - \varepsilon \cdot \langle p, q, l_j\langle v\rangle\rangle \upharpoonright p \\ & = & G \upharpoonright p - \langle p, q, l_j\langle v\rangle\rangle \upharpoonright p \\ & = & G \upharpoonright p - q!l_j(v). \end{array}$$

Since

$$G \upharpoonright p = T = q!\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.T_i\}_{i\in I},$$

by Definition 7.5.3,

$$G \upharpoonright p - q!l_j(v) = T_j\{v/x_j\}$$

which satisfies the 4th rule of $\mathfrak{E}$-coherence.

(b) Since $G \upharpoonright p - \widetilde{mv} \upharpoonright p$ in $\mathfrak{E}$ is defined (because $\mathfrak{E}$ is coherent), if $\widetilde{mv} \upharpoonright p \neq \varepsilon$, by Definition 7.5.2, assume

$$\widetilde{mv} \upharpoonright p = q_1!l^1(v^1) \cdot q_2!l^2(v^2) \cdot \ \ldots \ \cdot q_n!l^n(v^n),$$

where ^{i} is the upper index to denote the labels and values may be different, and $\forall\ k \in \{1..n\}, q_k!l^k(v^k) \neq \varepsilon$, is the kth output action of p.

Let $q_k!\{l^k_{j_k}(x^k_{j_k} : S^k_{j_k})\langle\langle A^k_{j_k}\rangle\rangle.T^k_{j_k}\}_{j_k\in I}$ be the kth left endpoint specification after the kth output action. Assume $G \upharpoonright p = T^0$, which is the endpoint specification at the beginning. Since $G \upharpoonright p - \widetilde{mv} \upharpoonright p = T$, by Definition 7.5.3, we have

$$\begin{array}{rl} & G \upharpoonright p - \widetilde{mv} \upharpoonright p \\ = & T^0 - q_1!l^1(v^1)\cdot q_2!l^2(v^2)\cdot \text{ ..... } \cdot q_n!l^n(v^n) \\ = & T^1_{j_1}\{v^1/x^1_{j_1}\} - q_2!l^2(v^2)\cdot \text{ ..... } \cdot q_n!l^n(v^n) \\ = & T^2_{j_2}\{v^2/x^2_{j_2}\} - q_3!l^3(v^3)\cdot \text{ ... } \cdot q_n!l^n(v^n) \\ = & \text{.....} \\ = & T^{n-1}_{j_{n-1}}\{v^{n-1}/x^{n-1}_{j_{n-1}}\} - q_n!l^n(v^n) \\ = & T^n_{j_n}\{v^n/x^n_{j_n}\} = T. \end{array}$$

where $l^k = l^k_{j_k}$. Thus inductively use Definition 7.5.3, we get

$$\begin{array}{rl} & G \upharpoonright p - (\widetilde{mv}\cdot\langle p,q,l_j\langle v\rangle\rangle) \upharpoonright p \\ = & T^0 - q_1!l^1(v^1)\cdot q_2!l^2(v^2)\cdot \text{ ..... } \cdot q_n!l^n(v^n)\cdot\langle p,q,l_j\langle v\rangle\rangle \upharpoonright p \\ = & T^1_{j_1}\{v^1/x^1_{j_1}\} - q_2!l^2(v^2)\cdot \text{ ..... } \cdot q_n!l^n(v^n)\cdot\langle p,q,l_j\langle v\rangle\rangle \upharpoonright p \\ = & T^2_{j_2}\{v^2/x^2_{j_2}\} - q_3!l^3(v^3)\cdot \text{ ... } \cdot q_n!l^n(v^n)\cdot\langle p,q,l_j\langle v\rangle\rangle \upharpoonright p \\ = & \text{.....} \\ = & T^{n-1}_{j_{n-1}}\{v^{n-1}/x^{n-1}_{j_{n-1}}\} - q_n!l^n(v^n)\cdot\langle p,q,l_j\langle v\rangle\rangle \upharpoonright p \\ = & T^n_{j_1}\{v^n/x^n_{j_n}\} - \langle p,q,l_j\langle v\rangle\rangle \upharpoonright p \\ = & T^n_{j_n}\{v^n/x^n_{j_n}\} - q!l_j(v). \end{array}$$

Since $T^n_{j_n}\{v^n/x^n_{j_n}\} = T = q!\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.T_i\}_{i\in I}$, by Definition 7.5.3, $T^n_{j_1}\{v^n/x^n_{j_n}\} - q!l_j(v) = T - q!l_j(v) = T_j\{v/x_j\}$, which satisfies the 4th rule of $\mathfrak{E}$-coherence.

In conclusion, $\mathfrak{E}'$ satisfies the 3rd rule of $\mathfrak{E}$-coherence such that

$$G \upharpoonright p - \widetilde{mv}\cdot\langle p,q,l_j\langle v\rangle\rangle \upharpoonright p = T_j\{v/x_j\},$$

$$T_j\{v/x_j\} \curvearrowright T_j\{v/x_j\}.$$

3. Assume $\mathfrak{E}$ is coherent, in $\mathfrak{E}$, $\exists\ s : G, s : \widetilde{mv}$.

   (a) Assume $s : \widetilde{mv}$ has no message sent from p to q. Since $\mathfrak{E}$ is coherent, in $\mathfrak{E}$

there exists T such that

$$\Delta(s[p]^\bullet) \curvearrowright q!\{l_i(x_i : S_i)\langle\!\langle A_i \rangle\!\rangle.T_i\}_{i\in I} = T,$$

and $G \upharpoonright p - \widetilde{mv} \upharpoonright p = T$. Lemma B.5.7 implies that

$$G \curvearrowright p \to q : \{l_i(x_i : S_i)\langle\!\langle A_i \rangle\!\rangle.G'_i\}_{i\in I}.$$

For the additional message $s : \widetilde{mv} \cdot \langle p, q, l_j\langle v\rangle\rangle$ in $\mathfrak{E}'$, since an output action $s[p, q]!l\langle v\rangle$ does not change the configuration of $s : G$ (since the interaction is not complete), it satisfies the 4th rule of $\mathfrak{E}$-coherence which says if $s : \widetilde{mv_1} \cdot s\langle p, q, l\langle v\rangle\rangle \cdot \widetilde{mv_2}$, $\widetilde{mv_1}$ does not suppress $s\langle p, q, l\langle v\rangle\rangle$, then $\exists\, G \curvearrowright p \to q : \{l_i(x_i : S_i)\langle\!\langle A_i \rangle\!\rangle.G'_i\}_{i\in I}$.

(b) Assume $s\langle p, q, l'\langle v'\rangle\rangle \in \widetilde{mv}$ and $G \curvearrowright p \to q : \{l'_i(x'_i : S'_i)\langle\!\langle A'_i \rangle\!\rangle.G'_i\}_{i\in I}$. As output $s[p, q]!l\langle v\rangle$ takes place, the configuration of G does not change and $G \not\curvearrowright p \to q : \{l_j(x_j : S_j)\langle\!\langle A_j \rangle\!\rangle.G'_j\}_{j\in J}$ because interaction $p \to q : \{l_j(x_j : S_j)\}_{j\in J}$ is suppressed by interaction $p \to q : \{l_i(x_i : S_i)\}_{i\in I}$. Thus it still satisfies the 4th rule of $\mathfrak{E}$-coherence.

Case [$\mathfrak{E}$-BCH/BCHN] As $\ell = s[p, q]?l_j(v)$, either v is a value or a name, according to rule [$\mathfrak{E}$-BCH], let

$$\mathfrak{E} = \mathfrak{E}_0, s : G_0, s[q]^\bullet : T_0, s : \widetilde{mv},$$

$$\widetilde{mv} = s\langle p, q, l_j\langle v\rangle\rangle \cdot \widetilde{mv'},$$

$$G_0 \curvearrowright p \to q : \{l_i(x_i : S_i)\langle\!\langle A_i \rangle\!\rangle.G_i\}_{i\in I} = G,$$

$$T_0 = G \upharpoonright q - \widetilde{mv} \upharpoonright q.$$

By [$\mathfrak{E}$-BCH], we get $\mathfrak{E}' = \mathfrak{E}_0, s : G'_0, s[q]^\bullet : T'_0, s : \widetilde{mv}$ where $G'_0 = G_j\{v/x_j\}$ and $T'_0 = (G_j \upharpoonright q)\{v/x_j\} - \widetilde{mv} \upharpoonright q$.

Similarly, we check if $\mathfrak{E}'$ satisfies three rules of $\mathfrak{E}$-coherence (Definition 7.5.6):

1. The only case that input action $s[p, q]?l_j(v)$ affects the existence of $s : G_0$ is when $G_0 = p \to q : \{l_i(x_i : S_i)\langle\!\langle A_i \rangle\!\rangle.\mathsf{end}\}, s : \langle p, q, l_j\langle v\rangle\rangle \in \mathfrak{E}$, thus $G'_0 \notin \mathfrak{E}'$ (because

of $G_0' = \mathsf{end}$ and $s : \emptyset$). Since $G_0 = p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.\mathsf{end}\}$ can only $\curvearrowright p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.\mathsf{end}\}$ and $\mathfrak{E}$ is coherent, $\Delta(s[q]^\bullet) = T_0 = p?\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.\mathsf{end}\} - \langle p, q, l_j\langle v\rangle\rangle \upharpoonright q = p?\{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.\mathsf{end}\}$, so that in $\mathfrak{E}'$, we have $G_0' = \mathsf{end}$, and $\Delta(s[q]^\bullet) = \mathsf{end}$, they finish all interactions and actions at the same time, thus satisfies the 2nd rule: if there is no such a global specification, then there is no corresponding endpoint specification.

2. For the 3rd rule of $\mathfrak{E}$-coherence, by rule [$\mathfrak{E}$-BCH], in $\mathfrak{E}'$, $s[q]^\bullet : T_0'$, $s : G_j\{v/x_j\}$, $T_0' = (G_j \upharpoonright q)\{v/x_j\} - \widetilde{mv} \upharpoonright q = (G_j \upharpoonright q)\{v/x_j\} - \widetilde{mv'} \upharpoonright q$, which means $T_0' \curvearrowright (G_j \upharpoonright q)\{v/x_j\} - \widetilde{mv'} \upharpoonright q$. Thus it satisfies the 3rd rule of $\mathfrak{E}$-coherence.

3. Since $\mathfrak{E}$ is coherent, every message specification $mv = s\langle p_k, q_k, l^k\langle v^k\rangle\rangle \in \widetilde{mv}$ implies that $G \curvearrowright p_k \to q_k : \{l_{i_k}^k({x_{i_k}}^k : {S_{i_k}}^k)\langle\langle A_{i_k}^k\rangle\rangle.G_{i_k}^k\}_{i_k \in I}$. Since we have message specifications $s : s\langle p, q, l\langle v\rangle\rangle \cdot \widetilde{mv'}$ in $\mathfrak{E}$ and $G \curvearrowright p \to q : \{l_i(x_i : S_i)\langle\langle A_i\rangle\rangle.G_i'\}_{i\in I}$, when message $\langle p, q, l\langle v\rangle\rangle$ is absorbed, $G \xrightarrow{s[p,q]?l(v)} G_j\{v/x_j\}$, and the rest interactions $p_k \to q_k : \{l_{i_k}^k({x_{i_k}}^k : {S_{i_k}}^k)\langle\langle A_{i_k}^k\rangle\rangle.G_{i_k}^k\}_{i_k \in I}$ are still in $G_j\{v/x_j\}$, which means they are still permutable to the top, therefore the 4th rule is satisfied.

Case [$\mathfrak{E}$-REQ-B] With the rule [$\mathfrak{E}$-REQ-B], when $\mathfrak{E} = \mathfrak{E}_0$ is coherent and $\ell = \overline{a}(s[p] : G)$, we have $s{:}G, s[p_i] : G \upharpoonright p_i$ where $\forall i \in I, p_i \in \mathit{role}(G)$, and $s : \emptyset$ in $\mathfrak{E}'$.

1. It is straightforward that $\mathfrak{E}'$ satisfies the 1st and 2nd rules of $\mathfrak{E}$-coherent.
2. Since there is no message in $s : \emptyset$, $\forall\, p \in \mathit{role}(G), G \upharpoonright p - \varepsilon \upharpoonright p = G \upharpoonright p = \Delta(s[p])$. Thus it satisfies the 3rd rule of $\mathfrak{E}$-coherence.
3. Since there is no message, the 4th rule is satisfied.

Other cases are trivial because they do not affect any rule of $\mathfrak{E}$-coherence. ■

Theorem 7.5.9 (session fidelity w.r.t. $\mathfrak{E}$). If $N \xrightarrow{\ell}_{\mathrm{g}} N'$, and $\mathfrak{E} \vdash N$, $\mathfrak{E}$ is coherent, then $\mathfrak{E} \xrightarrow{\ell}_{\mathrm{g}} \mathfrak{E}'$, and $\mathfrak{E}' \vdash N'$, $\mathfrak{E}'$ is coherent.

Proof. If $\mathfrak{E}$ is coherent, $\mathfrak{E} \xrightarrow{\ell}_{\mathrm{g}} \mathfrak{E}'$, then $\mathfrak{E}'$ is coherent has been proved in Proposition 7.5.7. Assume $\Theta_0, \Gamma_0, \Delta_0$ composes a global environment defined in $\mathfrak{E}$.

Case [sel/selN] For the case when $\ell = s[p,q]!l_j\langle v\rangle$, v is either a value or a name, assume

$$N = N_0 \parallel \mathcal{M}_s[P_s] \parallel H$$

by the fact $N \xrightarrow{\ell}_{\mathrm{g}} N'$ implying $\exists\ \mathcal{M}_s[P_s] \in N$ such that $\mathcal{M}_s[P_s] \xrightarrow{\ell} \mathcal{M}'_s[P'_s]$.

Since $\mathfrak{E} \vdash N$, with rules [$\mathfrak{E}$-CO] and [$\mathfrak{E}$-PAR], let $\mathfrak{E} = \Theta, \Gamma, \Delta_1, \Delta_2, \Delta_3$ such that

$$\begin{array}{rcl} \Theta, \Gamma, \Delta_1 & \vdash_{\mathrm{g}} & N_0, \\ \Theta, \Gamma, \Delta_2, \Delta_3 & \vdash_{\mathrm{g}} & \mathcal{M}_s[P_s] \parallel H \end{array}$$

By rule [$\mathfrak{E}$-PM3], $\Delta_3 = s : \widetilde{m}v$ so that $\Gamma, \Delta_3 \vdash_{\mathrm{g}} H$. By rules [$\mathfrak{E}$-M] and [Sel] (see Figure 7.4), let $\Delta_2 = \Delta_0, s[p]^\bullet : T$ where $T \curvearrowright q!\{l_i(x_i : S_i)\langle\!\langle A_i \rangle\!\rangle.T_i\}_{i\in I}$, $j \in I$, and $\Gamma \models A_j\{v/x_j\}$, so that

$$\Gamma, \Delta_2 \vdash_{\mathrm{g}} \mathcal{M}_s[P_s]$$

where

$$\mathcal{M}_s = \mathcal{M}_0, s[p]^\bullet : T,\ A_j\{v/x_j\} \downarrow \mathsf{true}.$$

based on the fact $\mathcal{M}_s \xrightarrow{\ell} \mathcal{M}'_s$. Therefore, rule [$\mathfrak{E}$-sel] says $\mathfrak{E} \xrightarrow{\ell}_{\mathrm{g}} \mathfrak{E}'$ because $\mathfrak{E}$ has $s[p]^\bullet : T$ which approves transition ℓ.

By rule [$\mathfrak{E}$-sel], $\mathfrak{E}' = \Theta, \Gamma, \Delta_1, \Delta'_2, \Delta'_3$ where $\Delta'_2 = \Delta_0, s[p]^\bullet : T_j\{v/x_j\}$ and $\Delta'_3 = s : \widetilde{m}v \cdot s\langle p, q, l_j\langle v\rangle\rangle$; and by rule [sel], $\mathcal{M}_s \xrightarrow{\ell} \mathcal{M}'_s$, thus $\mathcal{M}'_s = \mathcal{M}_0, s[p]^\bullet : T_j\{v/x_j\}$. With rule [$\mathfrak{E}$-PM3], they together imply that

$$\Theta, \Gamma, \Delta_1 \vdash_{\mathrm{g}} N_0,$$

$$\Theta, \Gamma, \Delta'_2, \Delta'_3 \vdash_{\mathrm{g}} \mathcal{M}'_s[P'_s] \mid H \cdot s\langle p, q, l_j\langle v\rangle\rangle$$

Thus, by applying rules [$\mathfrak{E}$-PAR] and [$\mathfrak{E}$-CO], we have $\mathfrak{E}' \vdash N_0 \parallel \mathcal{M}'_s[P'_s] \parallel H \cdot s\langle p, q, l_j\langle v\rangle\rangle = N'$.

Case [bch/bchN] For the case when $\ell = s[p,q]?l_j(v)$:

1. When v is a value, let $N = N_0 \parallel \mathcal{M}_r[P_r] \parallel s\langle p, q, l_j\langle v\rangle\rangle \cdot H$ by the fact $N \xrightarrow{\ell}_{\mathrm{g}} N'$ implying $\exists\ \mathcal{M}_r[P_r] \in N$, $\mathcal{M}_r[P_r] \xrightarrow{\ell} \mathcal{M}'_r[P'_r]$, where $\mathcal{M}_r[P_r]$ is a receiver. By rule [BRA] (see Figure 7.4), let

$$\mathcal{M}_r = \mathcal{M}_0, s[q]^\bullet : p?\{l_i(x_i : S_i)\langle\!\langle A_i\rangle\!\rangle.T_i\}_{i\in I},\ A_j\{v/x_j\} \downarrow \mathsf{true}.$$

Since $\mathfrak{E} \vdash N$, by rules [$\mathfrak{E}$-CO] and [$\mathfrak{E}$-PAR], let $\mathfrak{E} = \Theta, \Gamma, \Delta_1, \Delta_2, \Delta_3$ such that

$$\begin{array}{rcl} \Theta, \Gamma, \Delta_1 & \vdash_{\mathrm{g}} & N_0, \\ \Theta, \Gamma, \Delta_2, \Delta_3 & \vdash_{\mathrm{g}} & \mathcal{M}_r[P_r] \parallel s\langle p, q, l_j\langle v\rangle\rangle \cdot H \end{array}$$

By rule [$\mathfrak{E}$-PM3], let $\Delta_3 = s : s\langle p, q, l_j\langle v\rangle\rangle \cdot \widetilde{mv}$ so that $\Gamma, \Delta_3 \vdash_{\mathrm{g}} s\langle p, q, l_j\langle v\rangle\rangle \cdot H$.

Decompose $\widetilde{mv}$. Assume the messages sent from endpoint $s[q]$ are $s\langle q, p_k, l^k\langle v^k\rangle\rangle \in H$, such that for each k,

$$s : s\langle q, p_k, l^k\langle v^k\rangle\rangle \vdash_{\mathrm{g}} s\langle q, p_k, l^k\langle v^k\rangle\rangle,$$

$$s : s\langle q, p_k, l^k\langle v^k\rangle\rangle \in \Delta_3.$$

In $\mathfrak{E}$, we have $s : G \in \Theta$, with the decomposition of messages above, it should be in the form:

$$\begin{array}{rl} G = & q \to p_1 : \{l^1_{i_1}(x^1_{i_1} : S^1_{i_1})\langle\!\langle A^1_{i_1}\rangle\!\rangle.... \\ & q \to p_k : \{l^k_{i_k}(x^k_{i_k} : S^k_{i_k})\langle\!\langle A^k_{i_k}\rangle\!\rangle.... \\ & q \to p_m : \{l^m_{i_m}(x^m_{i_m} : S^m_{i_m})\langle\!\langle A^m_{i_m}\rangle\!\rangle.p \to q : \{l_i(x_i : S_i)\langle\!\langle A_i\rangle\!\rangle.G'_i\}_{i\in I}\}..\}..\} \end{array}$$

Since $\mathfrak{E}$ is coherent and $\mathfrak{E} \vdash N$, we have

$$\begin{array}{rl} G \upharpoonright q - (s\langle p, q, l_j\langle v\rangle\rangle \cdot \widetilde{mv}) \upharpoonright q & = G \upharpoonright q - \widetilde{mv} \upharpoonright q \\ & = p?\{l_i(x_i : S_i)\langle\!\langle A_i\rangle\!\rangle.G_i \upharpoonright q\}_{i\in I}. \end{array}$$

Since $\mathfrak{E}$ has $s : G, s[q]^\bullet : T$, and $s : s\langle p, q, l\langle v\rangle\rangle \cdot \widetilde{mv}$, where $G \curvearrowright p \to q : \{l_i(x_i : S_i)\langle\!\langle A_i\rangle\!\rangle.G_i\}_{i\in I}$, by rule [$\mathfrak{E}$-BCH], $\mathfrak{E}$ approves this action has transition $\mathfrak{E} \xrightarrow{\ell}_{\mathrm{g}} \mathfrak{E}'$.

Since $\mathcal{M}_r \xrightarrow{\ell} \mathcal{M}'_r$, by rule [BRA], $\mathcal{M}'_r = \mathcal{M}_0, s[q]^\bullet : T_j\{v/x_j\}$, and $H \in N'$.

Since $\mathfrak{E} \xrightarrow{\ell}_{\mathrm{g}} \mathfrak{E}'$, by rule [$\mathfrak{E}$-BCH], we have $\Theta', \Gamma, \Delta_1, \Delta_2', \Delta_3' \in \mathfrak{E}'$ where

$$\begin{array}{rcl} \Theta' & = & \Theta_0, G_j\{v/x_j\}, \\ \Delta_2' & = & \Delta_0, s[q]^\bullet : (G_j \upharpoonright q)\{v/x_j\} - \widetilde{m}v \upharpoonright q, \\ \Delta_3' & = & s : \widetilde{m}v \end{array}$$

where $(G_j \upharpoonright q)\{v/x_j\} - \widetilde{m}v \upharpoonright q = T_j\{v/x_j\}$ because

$$\begin{array}{rcl} (G_j \upharpoonright q - \widetilde{m}v \upharpoonright q)\{v/x_j\} & = & (G_j \upharpoonright q)\{v/x_j\} - (\widetilde{m}v \upharpoonright q)\{v/x_j\} \\ & = & (G_j \upharpoonright q)\{v/x_j\} - \widetilde{m}v \upharpoonright q \\ & = & T_j\{v/x_j\} \end{array}$$

so that

$$\Theta', \Gamma, \Delta_1 \vdash_{\mathrm{g}} N_0,$$

$$\Theta', \Gamma, \Delta_2', \Delta_3' \vdash \mathcal{M}_r'[P_r'] \parallel H.$$

By rule [$\mathfrak{E}$-PAR] and [$\mathfrak{E}$-CO], we have $\mathfrak{E}' \vdash N_0 \parallel \mathcal{M}_r'[P_r'] \parallel H = N'$.

2. When $\ell = s[p,q]?l_j(a)$ where a is a name, the proof is similar to the one of $\ell = s[p,q]?l_j(v)$, except setting $\Gamma \in \mathfrak{E}$ to be $\Gamma \vdash a : \mathtt{mode}(G[p])$ and $\Gamma \models A_j \downarrow \mathsf{true}$, other settings are the same. This setting does not affect transition according to rule [$\mathfrak{E}$-BCHN], so that we still have $\mathfrak{E} \xrightarrow{\ell}_{\mathrm{g}} \mathfrak{E}'$. After transition takes place, Γ does not change its configuration, which means that, in $\mathfrak{E}'$, for the additional $a : \mathtt{O}(G[p])$ in $\mathcal{M}_r'$, $\Gamma \vdash a : T[p]$. Thus $\mathfrak{E}' \vdash N'$.

Case [req-b] For the case when $\ell = \overline{a}(s[p_j] : G)$, for the session creation part is proved here, while for the request of invitation part is proved together with Case [req f]. Since $s \notin \mathsf{dom}(\mathfrak{E})$ is given and the fact $N \xrightarrow{\ell}_{\mathrm{g}} N'$ implying $\exists\ \mathcal{M}[P] \in N$ such that $\mathcal{M}[P] \xrightarrow{\ell} \mathcal{M}'[P']$, let

$$N = N_0 \parallel \mathcal{M}[P] \xrightarrow{\ell}_{\mathrm{g}} N_0 \parallel \mathcal{M}'[P] = N'.$$

Since $\mathfrak{E} \vdash N$, by rule [$\mathfrak{E}$-CO] and [$\mathfrak{E}$-PAR], let $\mathfrak{E} = \Theta, \Gamma, \Delta_1, \Delta_2$ such that

$$\begin{array}{rcl} \Theta, \Gamma, \Delta_1 & \vdash_{\mathrm{g}} & N_0, \\ \Theta, \Gamma, \Delta_2 & \vdash_{\mathrm{g}} & \mathcal{M}[P] \end{array}$$

Let $\Theta = \Theta_0$ and $\Delta_2 = \Delta_0, s \notin \mathsf{dom}(\Delta_0)$. By Lemma 7.2.6, $\mathcal{M} \xrightarrow{\ell} \mathcal{M}'$ and $s \notin \mathsf{dom}(\mathcal{M})$. By [$\mathfrak{E}$-REQ-B], $\mathfrak{E}$ approves this action and have transition $\mathfrak{E} \xrightarrow{\ell}_{\mathrm{g}} \mathfrak{E}'$; again, by [$\mathfrak{E}$-REQ-B], in $\mathfrak{E}'$, we have $\Theta', \Gamma, \Delta_1, \Delta_2' \in \mathfrak{E}'$ where $\Theta' = \Theta_0, s : G$, $\Delta_2' = \Delta_0, \{s[p_i] : G \upharpoonright p_i\}_{p_i \in role(G)}$ such that

$$\begin{array}{rcl} \Theta', \Gamma, \Delta_1 & \vdash_{\mathrm{g}} & N_0, \\ \Theta', \Gamma, \Delta_2' & \vdash_{\mathrm{g}} & \mathcal{M}'[P'] \end{array}$$

By rule [$\mathfrak{E}$-PAR] and [$\mathfrak{E}$-CO], we have $\mathfrak{E}' \vdash N_0 \parallel \mathcal{M}'[P']$.

Case [req-f] For the case when $\ell = \overline{a}\langle s[p] : G\rangle$ or $\ell = \overline{a}(s[p] : G)$, given the fact $N \xrightarrow{\ell}_{\mathrm{g}} N'$ implying $\exists\, \mathcal{M}[P] \in N$ such that $\mathcal{M}[P] \xrightarrow{\ell} \mathcal{M}'[P']$, let

$$N = N_0 \parallel \mathcal{M}[P] \parallel H \xrightarrow{\ell}_{\mathrm{g}} N_0 \parallel \mathcal{M}'[P] \parallel H \cdot \overline{a}\langle s[p] : G\rangle = N'.$$

Since $\mathfrak{E} \vdash N$, by rule [$\mathfrak{E}$-CO] and [$\mathfrak{E}$-PAR], let $\mathfrak{E} = \Theta, \Gamma, \Delta_1, \Delta_2, \Delta_3$ such that

$$\begin{array}{rcl} \Theta, \Gamma, \Delta_1 & \vdash_{\mathrm{g}} & N_0, \\ \Theta, \Gamma, \Delta_2, \Delta_3 & \vdash_{\mathrm{g}} & \mathcal{M}[P] \parallel H \end{array}$$

where $\Gamma = \Gamma_0, a : \mathtt{om}(G[p])$ $\Delta_2 = \Delta_0, s[p] : T$, and $\Delta_3 = s : \widetilde{mv}$ such that $\Delta_2 \vdash_{\mathrm{g}} \mathcal{M}[P], \Delta_3 \vdash_{\mathrm{g}} H$. By rule [$\mathfrak{E}$-M], when given $\mathcal{M} = \mathcal{M}_0, a : \mathtt{om}(G[p]), s[p] : T, a \notin \mathsf{dom}(\mathcal{M}_0)$ by rule [REQ-F]. Since $\mathfrak{E}$ is coherent, the existence of $a : \mathtt{om}(G[p])$ implies the existence of $a : \mathtt{im}(G[p])$ in $\mathfrak{E}$.

By rule [$\mathfrak{E}$-REQ-F], $\mathfrak{E}$ allows this action and have transition $\mathfrak{E} \xrightarrow{\ell}_{\mathrm{g}} \mathfrak{E}'$, again, by rule [$\mathfrak{E}$-REQ-F], $\mathfrak{E} = \mathfrak{E}'$; therefore, with rule [$\mathfrak{E}$-CO], [$\mathfrak{E}$-PM1], and given $\mathcal{M}' = \mathcal{M}_0, a : \mathtt{om}(G[p])$, we have we have $\mathfrak{E}' \vdash N_0 \mid \mathcal{M}'[P'] \parallel H \cdot \overline{a}\langle s[p] : G\rangle$.

Case [acc-b/f] For case $\ell = a(s[p] : G)$ or $\ell = a\langle s[p] : G\rangle$, given the fact $N \xrightarrow{\ell}_{\mathrm{g}} N'$, $\exists\, \mathcal{M}[P] \in N$ such that $\mathcal{M}[P] \xrightarrow{\ell} \mathcal{M}'[P']$, let $N = N_0 \parallel \mathcal{M}[P] \parallel \overline{a}\langle s[p] : G\rangle \cdot H \xrightarrow{\ell}_{\mathrm{g}} N_0 \parallel \mathcal{M}'[P'] \parallel H = N'$.

Since $\mathfrak{E} \vdash N$, by rule [$\mathfrak{E}$-CO] and [$\mathfrak{E}$-PAR], let $\mathfrak{E} = \Theta, \Gamma, \Delta_1, \Delta_2, \Delta_3$ such that

$$\begin{array}{rcl} \Theta, \Gamma, \Delta_1 & \vdash_{\mathrm{g}} & N_0, \\ \Theta, \Gamma, \Delta_2, \Delta_3 & \vdash_{\mathrm{g}} & \mathcal{M}[P] \parallel \overline{a}\langle s[p] : G\rangle \cdot H \end{array}$$

where $\Delta_3 = s : \widetilde{mv}, \Delta_3 \vdash_{\mathrm{g}} H$. By rule [$\mathfrak{E}$-M], [$\mathfrak{E}$-PM1], and given $\mathcal{M} = \mathcal{M}_0, a : \mathtt{im}(G[p])$ by rule [ACC-B/F], $\Gamma = \Gamma_0, a : \mathtt{im}(G[p])$, and $\Delta_2 = \Delta_0, s[p] : T$.

By rule [$\mathfrak{E}$-ACC], $\mathfrak{E}$ approves this action and has transition $\mathfrak{E} \xrightarrow{\ell}_{\mathrm{g}} \mathfrak{E}'$. By rule [$\mathfrak{E}$-ACC], $\mathfrak{E} = \mathfrak{E}'$, and $\mathcal{M}' = \mathcal{M}_0, a : \mathtt{im}(G[p]), s[p] : T$. Therefore, with rule [$\mathfrak{E}$-CO], we have $\mathfrak{E}' \vdash N_0 \parallel \mathcal{M}'[P']$.

Case [new a] For case $\ell = \mathsf{new}\ a : \mathtt{im}(G[p])$, since the fact $N \xrightarrow{\ell}_{\mathrm{g}} N'$, $\exists\ \mathcal{M}[P] \in N$ such that $\mathcal{M}[P] \xrightarrow{\ell} \mathcal{M}'[P']$. Let $N = N_0 \parallel \mathcal{M}[P] \xrightarrow{\ell}_{\mathrm{g}} N_0 \parallel \mathcal{M}'[P] = N'$.

Since $\mathfrak{E} \vdash N$, by rule [$\mathfrak{E}$-CO] and [$\mathfrak{E}$-PAR], let $\mathfrak{E} = \Theta, \Gamma, \Delta$ such that

$$\begin{array}{rcl} \Theta, \Gamma, \Delta & \vdash_{\mathrm{g}} & N_0, \\ \Gamma, \Delta & \vdash_{\mathrm{g}} & \mathcal{M}[P] \end{array}$$

where $\Gamma = \Gamma_0$ and $a \notin \Gamma_0$.

By rule [$\mathfrak{E}$-NEW A], $\mathfrak{E}$ approves this transition as $\mathfrak{E} \xrightarrow{\ell}_{\mathrm{g}} \mathfrak{E}'$. By rule [$\mathfrak{E}$-NEW A], $\Theta, \Gamma', \Delta \in \mathfrak{E}'$, $\Gamma' = \Gamma_0, a : \mathtt{im}(G[p])$. We have $\mathcal{M}' = \mathcal{M}, a : \mathtt{im}(G[p])$ by rule [NEW A]. By rule [$\mathfrak{E}$-CO] and [$\mathfrak{E}$-M], we have $\mathfrak{E}' \vdash N_0 \parallel \mathcal{M}'[P']$.

Case [join] For case $\ell = \mathsf{join}(s[p])$, the proof is trivial. Let

$$N = N_0 \parallel \mathcal{M}[P] \xrightarrow{\ell}_{\mathrm{g}} N_0 \parallel \mathcal{M}'[P] = N'.$$

Since the fact $N \xrightarrow{\ell}_{\mathrm{g}} N'$, it implies $\exists\ \mathcal{M}[P] \in N, \mathcal{M}[P] \xrightarrow{\ell} \mathcal{M}'[P']$.

Since $\mathfrak{E} \vdash N$, by rules [$\mathfrak{E}$-CO] and [$\mathfrak{E}$-PAR], let $\mathfrak{E} = \Theta, \Gamma, \Delta_1, \Delta_2$ such that

$$\begin{array}{rcl} \Theta, \Gamma, \Delta_1 & \vdash_{\mathrm{g}} & N_0, \\ \Theta, \Gamma, \Delta_2 & \vdash_{\mathrm{g}} & \mathcal{M}[P] \end{array}$$

where $\Delta_2 = \Delta_0, s[p] : T$ by rule [$\mathfrak{E}$-M].

By rule [$\mathfrak{E}$-JOIN], $\mathfrak{E}$ approves this action and have transition $\mathfrak{E} \xrightarrow{\ell}_{\mathrm{g}} \mathfrak{E}'$; again, by rule [$\mathfrak{E}$-JOIN], $\Theta, \Gamma, \Delta_2' \in \mathfrak{E}'$, $\Delta_2' = \Delta_0, s[p]^\bullet : T$.

Given $\mathcal{M}' = \mathcal{M}_0, s[p]^\bullet : T$ by rule [JOIN], and together with rule [$\mathfrak{E}$-CO] and [$\mathfrak{E}$-PAR], we have $\mathfrak{E}' \vdash N_0 \parallel \mathcal{M}'[P']$, which is denoted as N'.

Case [tau] as $\ell = \tau$, it is trivial.■

B. APPENDIX FOR DYNAMIC ASYNCHRONOUSLY MONITORING

APPENDIX C

Appendix for Specifying Stateful Asynchronous Properties for Distributed Programs

C.1 Auxiliary Theorems and Proofs for Strongest Specifications

Let $\Theta_{\mathsf{sync}}, \Theta_{\mathsf{async}}$ and Θ_{ass} be the local specifications projected from global stateful protocols $G_{\mathsf{sync}}, G_{\mathsf{async}}$ and G_{assign}, respectively (refer to Example 8.1.2 and Example 8.1.3), on the role S (i.e. server). They are defined as follows:

$$\begin{array}{rcl}
\Theta_{\mathsf{sync}} & = & \langle \mathtt{ser} : \mathtt{I}(G_{\mathsf{sync}}[S]); \\
& & \{s_j[S] : C?\mathrm{req}(\varepsilon)\langle\!\langle \mathsf{true}\ ; \varepsilon\rangle\!\rangle.C!\mathrm{ans}(x : \mathsf{int})\langle\!\langle x = \mathbf{c}\ ;\ \mathbf{c} := \mathbf{c} + 1\rangle\!\rangle\}_{j\in J}; \\
& & \mathbf{c} \mapsto \mathsf{ini}\rangle \\
\\
\Theta_{\mathsf{async}} & = & \langle \mathtt{ser} : \mathtt{I}(G_{\mathsf{async}}[S])); \\
& & \{s_j[S] : C?\mathrm{req}(\varepsilon)\langle\!\langle \mathsf{true}\ ; \varepsilon\rangle\!\rangle.C!\mathrm{ans}(x : \mathsf{int})\langle\!\langle x \notin \mathbf{c};\ \mathbf{c} := \mathbf{c} \cup \{x\}\rangle\!\rangle\}_{j\in J}; \\
& & \mathbf{c} \mapsto \{\ \ \}\rangle \\
\\
\Theta_{\mathsf{ass}} & = & \langle \mathtt{ser} : \mathtt{I}(G_{\mathsf{assign}}[S]); \\
& & \{s_j[S] : C?\mathrm{req}(\varepsilon)\langle\!\langle \mathsf{true}\ ;\ \mathbf{c}' := \mathbf{c}' + 1, \mathbf{log} := \mathbf{log} \cup \{\mathbf{c}'\}\rangle\!\rangle. \\
& & \qquad\qquad C!\mathrm{ans}(x : \mathsf{int})\langle\!\langle x \in \mathbf{log}\ ;\ \mathbf{log} := \mathbf{log} \setminus \{x\}\rangle\!\rangle\}_{j\in J}\ ; \\
& & \mathbf{c}' \mapsto \mathsf{ini}', \mathbf{log} \mapsto \{\ \ \}\rangle
\end{array}$$

where J is the set of indexes of sessions.

C. APPENDIX FOR SPECIFYING STATEFUL ASYNCHRONOUS PROPERTIES FOR DISTRIBUTED PROGRAMS

Lemma C.1.1. For any $\mathtt{s}' \in \mathtt{prefix}(\mathtt{s})$, $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{sync}})$, we always have

$$\mathsf{num}(\mathtt{O}(\mathtt{s}')) \leq \mathsf{num}(\mathtt{I}(\mathtt{s}'))$$

Proof. Simply by the definition of Θ_{sync} which specifies that inputs should always happen before outputs in the same session. Thus, in every possible prefix of $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{sync}})$, the number of inputs should be more than the number of outputs.■

Lemma C.1.2. Assume $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{sync}})$ and $\mathbf{c} \in \mathtt{field}(\Theta_{\mathsf{sync}})$. Assume $\mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \emptyset) = \mathsf{ini}$. Then $\forall \mathtt{s}' \in \mathtt{prefix}(\mathtt{s})$, we always have

$$\mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}') = \mathsf{num}(\mathtt{O}(\mathtt{s}')) + \mathsf{ini}$$

Proof. By induction, when $\mathsf{num}(\mathtt{O}(\mathtt{s}')) = 0$, which means there is no output action, $0 + \mathsf{ini} = \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}')$. It is correct because no output action in $\mathtt{s}'$ means no update $\mathbf{c} := \mathbf{c} + 1$ applied according to Θ_{sync}.

Let Θ'_{sync} be the configuration from $\Theta_{\mathsf{sync}} \xrightarrow{\mathtt{s}'} \Theta'_{\mathsf{sync}}$. Assume $\mathsf{num}(\mathtt{O}(\mathtt{s}')) = k$ so that $\mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}') = k + \mathsf{ini}$, which means $\mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}') = \mathtt{val}(\mathbf{c}, \Theta'_{\mathsf{sync}}) = k + \mathsf{ini}$. For $\mathtt{s}' \cdot \ell$:

1. When ℓ is an input, $\mathsf{num}(\mathtt{O}(\mathtt{s}' \cdot \ell)) = \mathsf{num}(\mathtt{O}(\mathtt{s}')) = k$, and according to Θ_{sync}, state $\mathbf{c}$ is *not* updated, which means

   $$\mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}' \cdot \ell) = \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}') = k + \mathsf{ini} = \mathsf{num}(\mathtt{O}(\mathtt{s}')) + \mathsf{ini}.$$

2. When ℓ is an output, $\mathsf{num}(\mathtt{O}(\mathtt{s}' \cdot \ell)) = \mathsf{num}(\mathtt{O}(\mathtt{s}')) + 1 = k + 1$. According to Θ_{sync}, state $\mathbf{c}$ is updated by $\mathbf{c} := \mathbf{c} + 1$. Therefore, let Θ''_{sync} be the configuration from $\Theta'_{\mathsf{sync}} \xrightarrow{\ell} \Theta''_{\mathsf{sync}}$, we have

   $$\mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}' \cdot \ell) = \mathtt{val}(\mathbf{c}, \Theta''_{\mathsf{sync}}) = \mathtt{val}(\mathbf{c}, \Theta'_{\mathsf{sync}}) + 1 = (k + \mathsf{ini}) + 1$$

   which means

   $$\mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}' \cdot \ell) = (k + 1) + \mathsf{ini} = \mathsf{num}(\mathtt{O}(\mathtt{s}' \cdot \ell)) + \mathsf{ini}.$$

By induction, the statement is proved.■

Lemma C.1.3. Assume $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{sync}})$ and $\mathbf{c} \in \mathtt{field}(\Theta_{\mathsf{sync}})$. For any $\mathtt{s}' \in \mathtt{prefix}(\mathtt{s})$, $\mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}') \leq \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s})$.

Proof. According to Θ_{sync}, $\mathbf{c}$ is updated by $\mathbf{c} := \mathbf{c} + 1$ when an output happens, which means the value of $\mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}'')$ depends on the number of outputs happening in $\mathtt{s}''$. Because the number of outputs in $\mathtt{s}'$ is smaller than or equal to the number of outpus in $\mathtt{s}$, we always have $\mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}') \leq \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s})$. ■

Lemma C.1.4. Assume $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{sync}})$ and $\mathbf{c} \in \mathtt{field}(\Theta_{\mathsf{sync}})$. Then for any $\ell \in \mathtt{O}(\mathtt{s}'), \mathtt{s}' \in \mathtt{prefix}(\mathtt{s})$, $\mathtt{v}(\ell) < \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}')$.

Proof. By Lemma C.1.3, we always have $\mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}_0) \leq \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}')$ whenever $\mathtt{s}_0$ is a prefix of $\mathtt{s}'$ because $\mathsf{num}(\mathtt{s}_0) \leq \mathsf{num}(\mathtt{s}')$.

Let ℓ^* be the last output in $\mathtt{s}' = \mathtt{s}_0 \cdot \ell^* \cdot \mathtt{s}_1$, which means there is no output in $\mathtt{s}_1$. Then, according to Θ_{sync}, $\mathtt{v}(\ell^*) = \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}_0)$. This fact implies two things:

(a) For any other $\ell \in \mathtt{O}(\mathtt{s}')$, we know $\ell \in \mathtt{s}_0 = \mathtt{s}_0' \cdot \ell \cdot \mathtt{s}_0''$ because ℓ^* is the last output. According to Θ_{sync}, $\mathtt{v}(\ell) = \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}_0') \leq \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}_0) = \mathtt{v}(\ell^*)$.

(b) $\mathtt{v}(\ell^*) = \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}_0) < \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}_0 \cdot \ell^*) = \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}')$, so that

$$\mathtt{v}(\ell^*) < \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}')$$

Together with (a) and (b), for any $\ell \in \mathtt{s}', \mathtt{s}' \in \mathtt{prefix}(\mathtt{s})$, $\mathtt{v}(\ell) < \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}')$. ■

Lemma C.1.5. Assume $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{sync}})$ and $\mathbf{c} \in \mathtt{field}(\Theta_{\mathsf{sync}})$. Assume $\mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \emptyset) = \mathsf{ini}$. $\forall \mathtt{s}' \in \mathtt{prefix}(\mathtt{s})$, we have

$$\mathsf{ini} \leq \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}') \leq \mathsf{num}(\mathtt{I}(\mathtt{s}')) + \mathsf{ini}.$$

Proof. By Lemma C.1.2, $\mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}') = \mathsf{num}(\mathtt{O}(\mathtt{s}'))+\mathsf{ini}$ so that $\mathsf{ini} \leq \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}')$. Moreover, by Lemma C.1.1 which states that $\mathsf{num}(\mathtt{O}(\mathtt{s}')) \leq \mathsf{num}(\mathtt{I}(\mathtt{s}'))$, we have

$$\forall \mathtt{s}' \in \mathtt{prefix}(\mathtt{s}), \mathsf{ini} \leq \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}') = \mathsf{num}(\mathtt{O}(\mathtt{s}')) + \mathsf{ini} \leq \mathsf{num}(\mathtt{I}(\mathtt{s}')) + \mathsf{ini}.$$

Then the proof is done. ■

Lemma C.1.6. Assume $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{sync}})$, $\mathbf{c} \in \mathtt{field}(\Theta_{\mathsf{sync}})$. Assume $\mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \emptyset) = \mathsf{ini}$. $\forall \ell \in \mathtt{O}(\mathtt{s}'), \mathtt{s}' \in \mathtt{prefix}(\mathtt{s})$, we always have $\mathsf{ini} \leq \mathtt{v}(\ell) < \mathsf{num}(\mathtt{I}(\mathtt{s}')) + \mathsf{ini}$.

Proof. Immediately from Lemmas C.1.4 and C.1.5. ■

Lemma C.1.7. Assume $\mathtt{s} = \mathtt{s}_1 \cdot \ell$ and $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{sync}})$. If ℓ can permute with $\mathtt{s}_1$ such that $\mathtt{s}_1 \cdot \ell \curvearrowright \mathtt{s}_1' \cdot \ell'$, then $\mathsf{num}(\mathtt{I}(\mathtt{s}_1')) > \mathsf{num}(\mathtt{O}(\mathtt{s}_1'))$.

Proof. When $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{sync}})$, only when ℓ is input then it can permute with an output or an input in $\mathtt{s}_1$. Therefore, $\mathsf{num}(\mathtt{I}(\mathtt{s}_1')) > \mathsf{num}(\mathtt{O}(\mathtt{s}_1'))$. ■

Lemma 8.5.13. Assume $\mathbf{c} \in \mathtt{field}(\Theta_{\mathsf{sync}})$ and $\mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \emptyset) = \mathsf{ini}$. Assume every session s_i guided by G_{sync} has been established. Θ_{sync} is defined as below:

$$\begin{array}{rcl} \Theta_{\mathsf{sync}} & = & \langle \mathtt{ser} : \mathtt{I}(G_{\mathsf{sync}}[S]) \ ; \\ & & \{s_i[S] : C?\mathrm{req}(\varepsilon)\langle \mathsf{true} \ ; \varepsilon\rangle . C!\mathrm{ans}(x : \mathsf{int})\langle x = \mathbf{c} \ ; \ \mathbf{c} := \mathbf{c} + 1\rangle\}_{i \in I} \ ; \\ & & \mathbf{c} \mapsto \mathsf{ini}\rangle \end{array}$$

where I is the index of sessions. If $\mathtt{s}' \in \mathtt{trace}(\Theta_{\mathsf{sync}})$, then for any $\mathtt{s}$ resulting from permutations $\mathtt{s}' \curvearrowright \mathtt{s}$, $\mathtt{s}$ satisfies the followings:

cond 1. If session $s \in \mathtt{ses}(\mathtt{s})$, then $s \in \mathtt{ses}(\Theta_{\mathsf{sync}})$.

cond 2. $\forall \mathtt{s}'' \in \mathtt{prefix}(\mathtt{s})$, $\mathsf{num}(\mathtt{I}(\mathtt{s}'')) \geq \mathsf{num}(\mathtt{O}(\mathtt{s}''))$;

cond 3. If $\mathtt{s}_0 \cdot \ell \in \mathtt{s} \ \cap \ \ell = s[S, B]!\mathrm{ans}(v)$, then $\exists \ \ell' \in \mathtt{s}_0$, $\ell' = s[B, S]?\mathrm{req}(\varepsilon)$.

cond 4. $\forall \ell, \ell' \in \mathtt{O}(\mathtt{s})$, $\ell \neq \ell'$, then $\mathtt{v}(\ell) \neq \mathtt{v}(\ell')$;

cond 5. $\forall \ell \in \mathtt{O}(\mathtt{s}''), \mathtt{s}'' \in \mathtt{prefix}(\mathtt{s}), \mathsf{ini} \leq \mathtt{v}(\ell) < \mathsf{num}(\mathtt{I}(\mathtt{s}'')) + \mathsf{ini}$.

Proof. First of all, for any $\mathtt{s}' \in \mathtt{trace}(\Theta_{\mathsf{sync}})$, $\mathtt{s}'$ satisfies all these conditions: condition 0 is satisfied because $\mathtt{s}'$ is valid according to Θ_{sync} which means, for any $s \in \mathtt{ses}(\mathtt{s}')$, Θ_{sync} should include this session so that it can verify the actions involving in this session. Conditions 1,2, 3 are satisfied simply by the definition of Θ_{sync} and condition 4 is satisfied by Lemmas C.1.6. So the main proof below is to show, when $\mathtt{s}'$ satisfies these five conditions and $\mathtt{s}' \curvearrowright \mathtt{s}$, $\mathtt{s}$ satisfies all these conditions.

For cond 1. Since any permutation of $\mathtt{s}'$ does not add or subtract any session from $\mathtt{s}'$, any trace $\mathtt{s}$ permuted from $\mathtt{s}' \curvearrowright \mathtt{s}$ has the sessions as same as $\mathtt{s}'$ does. We have any $s \in \mathtt{ses}(\mathtt{s})$ implies $s \in \mathtt{ses}(\mathtt{s}')$, which again implies $s \in \mathtt{ses}(\Theta_{\mathsf{sync}})$.

For cond 2. By the permutation rules defined in Definition 8.4.12,

(a) for any action sequence $\ell{\cdot}\ell' \in \mathtt{s}'$, where ℓ is output and ℓ' is input, if they are permutable and are permuted by doing $\ell{\cdot}\ell' \curvearrowright \ell'{\cdot}\ell$, so that it makes $\mathtt{s}' \curvearrowright \mathtt{s}$, $\mathtt{s}$ still satisfies cond 2 because the number of inputs in every $\mathtt{s}'' \in \mathtt{prefix}(\mathtt{s})$ is more than the number of inputs in every $\mathtt{s}''' \in \mathtt{prefix}(\mathtt{s}')$.

(b) for any action sequence $\ell{\cdot}\ell' \in \mathtt{s}'$, which are both inputs (resp. outputs), after permutations $\ell{\cdot}\ell' \curvearrowright \ell'{\cdot}\ell$, which leads to $\mathtt{s}' \curvearrowright \mathtt{s}$, the number of inputs or the number of outputs in every $\mathtt{s}'' \in \mathtt{prefix}(\mathtt{s})$ is as same as that in every $\mathtt{s}''' \in \mathtt{prefix}(\mathtt{s}')$.

For cond 3. Based on $\mathtt{s}' \in \mathtt{trace}(\Theta_{\mathsf{sync}})$, since the permutation of $\mathtt{s}' \curvearrowright \mathtt{s}$ only moves input actions ahead of output actions in $\mathtt{s}$, $\mathtt{s}$ still satisfies cond 3.

For cond 4. Since $\mathtt{s}' \in \mathtt{trace}(\Theta_{\mathsf{sync}})$, every outputted value of a trace in $\mathtt{s}'$ is unique. Then for every $\mathtt{s}$ permuted from $\mathtt{s}' \curvearrowright \mathtt{s}$, every outputted value of $\mathtt{s}$ is still unique.

For cond 5. Based on $\mathtt{s}' \in \mathtt{trace}(\Theta_{\mathsf{sync}})$ and $\mathtt{s}'$ satisfies cond 5, when $\mathtt{s}$ is the result from permutation $\mathtt{s}' \curvearrowright \mathtt{s}$, because the valid permutations only make inputs move ahead of outputs, or two inputs (resp. two outputs) permute to each other, overall the permutations only increase the number of input actions ahead of output actions in $\mathtt{s}$, while the value of any output action is not changed. By Lemma C.1.7, $\mathtt{s}$ satisfies cond 5. ■

Remark C.1.8. Based on Lemma 8.5.13, some trace $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{async}})$ may not satisfy these conditions because $\mathtt{trace}(\Theta_{\mathsf{async}})$ is much bigger than the set

$$\{\mathtt{s} \mid \mathtt{s}' \in \mathtt{trace}(\Theta_{\mathsf{sync}}), \mathtt{s}' \curvearrowright \mathtt{s}\}.$$

Definition C.1.9 (contiguous series). For a set $\{v_0, ..., v_n\}$ of numbers, where v_0 is the minimum element and v_n is the maximum element. If , except v_n, any $v \in \{v_0, ..., v_n\}$, $\exists v' \in \{v_0, ..., v_n\}$ such that $v' = v + 1$, then the set is called a contiguous series.

Lemma C.1.10. The states $\mathbf{c}'$ and **log** in Θ_{ass} repectively have the following attributes:

1. For state $\mathbf{c}'$, for any $\mathtt{s}$,

(a) when ℓ is an input, $\texttt{current}(\mathbf{c}', \Theta_{\mathsf{ass}}, \mathtt{s}\cdot\ell) = \texttt{current}(\mathbf{c}', \Theta_{\mathsf{ass}}, \mathtt{s}) + 1$.

(b) when ℓ is an output, $\texttt{current}(\mathbf{c}', \Theta_{\mathsf{ass}}, \mathtt{s}\cdot\ell) = \texttt{current}(\mathbf{c}', \Theta_{\mathsf{ass}}, \mathtt{s})$.

2. For state **log**, for any $\mathtt{s}$,

(a) when ℓ is an input, $\texttt{current}(\mathbf{log}, \Theta_{\mathsf{ass}}, \mathtt{s}\ell) = \texttt{current}(\mathbf{log}, \Theta_{\mathsf{ass}}, \mathtt{s}) \cup \{\texttt{current}(\mathbf{c}', \Theta_{\mathsf{ass}}, \mathtt{s}\cdot\ell)\}$.

(b) when ℓ is an output, $\texttt{current}(\mathbf{c}', \Theta_{\mathsf{ass}}, \mathtt{s}\cdot\ell) = \texttt{current}(\mathbf{c}', \Theta_{\mathsf{ass}}, \mathtt{s}) \setminus \{\mathtt{v}(\ell)\}$.

Proof. It is proved directly from the mechanism of Θ_{ass}.

Proposition 8.5.14. Assume $\mathbf{c}', \mathbf{log} \in \texttt{field}(\Theta_{\mathsf{ass}})$and $\texttt{current}(\mathbf{c}', \Theta_{\mathsf{ass}}, \emptyset) = \mathsf{ini}' = \mathsf{ini} - 1$, $\texttt{current}(\mathbf{log}, \Theta_{\mathsf{ass}}, \emptyset) = \{\ \}$. Assume every session s_i guided by G_{assign} has been established. Θ_{ass} is defined below:

$$
\begin{array}{rcl}
\Theta_{\mathsf{ass}} & = & \langle \mathtt{ser} : \mathtt{I}(G_{\mathsf{assign}}[S]) \ ; \\
& & \{s_i[S] : C?\mathrm{req}(\varepsilon)\langle \mathsf{true} \ ; \ \mathbf{c}' := \mathbf{c}' + 1, \mathbf{log} := \mathbf{log} \cup \{\mathbf{c}'\}\rangle . \\
& & \qquad C!\mathrm{ans}(x : \mathsf{int})\langle x \in \mathbf{log} \ ; \ \mathbf{log} := \mathbf{log} \setminus \{x\}\rangle\}_{i \in I} \ ; \\
& & \mathbf{c}' \mapsto \mathsf{ini}', \mathbf{log} \mapsto \{\ \}\rangle
\end{array}
$$

where I is the index of sessions. $\mathtt{s}$ satisfies all conditions listed in Lemma 8.5.13, *if and only if* $\mathtt{s} \in \texttt{trace}(\Theta_{\mathsf{ass}})$.

Proof. (For $\Rightarrow$): Assume $\mathtt{s}$ satisfies all conditions listed in Lemma 8.5.13. We only need to make sure that every output action in $\mathtt{s}$ always satisfies the predicate at output obligation (i.e. $x \in \mathbf{log}$). The reasonings are as follows:

(1) Based on cond 1, 2, and 3, for any session $s \in \texttt{ses}(\mathtt{s})$, its actions are either $\mathtt{s}' \cdot s[C, S]?\mathrm{req}(\varepsilon) \cdot \mathtt{s}'' \cdot s[S, C]!\mathrm{ans}(v)$ or $\mathtt{s}' \cdot s[C, S]?\mathrm{req}(\varepsilon)$ where $s[S, C]!\mathrm{ans}(v) \notin \mathtt{s}'$. Thus the sequence of actions satisfy that Θ_{ass} defines, for a session, its input action should happen before its ouput.

(2) Although an input is anyway valid because of the predicate true at input obligation, with the update $\mathbf{c}' := \mathbf{c}' + 1$ and $\mathbf{log} := \mathbf{log} \cup \{\mathbf{c}'\}$, it affects the elements of set **log** and thus may affect the judgement of predicate $x \in \mathbf{log}$ when replacing x with the value of the upcoming output action.

For the effects caused by inputs the statement we have for $\mathtt{s}$ satisfying the five conditions:

Claim. $\mathtt{current}(\mathbf{c}', \Theta_{\mathsf{ass}}, \mathtt{s}) = \mathsf{num}(\mathtt{I}(\mathtt{s})) + \mathsf{ini}'$.

This is because $\mathbf{c}'$ is a counter at input obligation whose predicate is true, any output action does not update $\mathbf{c}'$ and, for any $\mathtt{s}$, $\mathbf{c}'$ will be updated whenever an input happens. Note *it does not matter* whether $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{ass}})$ or not (at this point this has not been proved).

This statement can be proved similarly as the proof in Lemma C.1.2. In Lemma *C*.1.2, the statement is $\mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}) = \mathsf{num}(\mathtt{O}(\mathtt{s})) + \mathsf{ini}'$ because $\mathbf{c}$ is a counter at output obligation in Θ_{sync}, while $\mathbf{c}'$ is a counter at input obligation in Θ_{ass}. Remember, the predicate in Θ_{ass} for any input is true and any output action cannot update $\mathbf{c}'$. We only need to replace every $\mathbf{c}$ with $\mathbf{c}'$ and Θ_{sync} with Θ_{ass}, and change $\mathsf{num}(\mathtt{O}(\mathtt{s}))$ to $\mathsf{num}(\mathtt{I}(\mathtt{s}))$, other steps of the proof are the same.

(3) For any output action ℓ in $\mathtt{s}$, since more than one inputs may have happened before ℓ according to cond 2 and cond 3, to satisfy the predicate $x \in \mathbf{log}$ when replacing x with $\mathtt{v}(\ell)$, the followings should be considered:

For any $\mathtt{s}$, let $\mathtt{s} = \mathtt{s}_0 \cdot \ell \cdot \mathtt{s}_1$, where $\mathtt{s}_0$ only has inputs so that ℓ is the first output.

(a) Up to $\mathtt{s}_0$, Θ_{ass} accepts $\mathtt{s}_0$ because it only has inputs. According to updates $\mathbf{c}' := \mathbf{c}' + 1$ and $\mathbf{log} := \mathbf{log} \cup \{\mathbf{c}'\}$ at input obligation, where $\mathbf{c}'$ has initial value ini', and $\mathbf{log}$ has initial value $\{\ \}$, the minimum (first) element in $\mathbf{log}$ is $\mathsf{ini}' + 1 = \mathsf{ini}$, we therefore have,

$$\mathtt{current}(\mathbf{log}, \Theta_{\mathsf{ass}}, \mathtt{s}_0) = \{\mathsf{ini}'+1, ..., \mathsf{ini}'+\mathsf{num}(\mathtt{s}_0)\} = \{\mathsf{ini}, ..., (\mathsf{ini}-1)+\mathsf{num}(\mathtt{s}_0)\}$$

Thus for any $\mathtt{s}'' \in \mathtt{prefix}(\mathtt{s}_0)$,

$$\mathtt{current}(\mathbf{c}', \Theta_{\mathsf{ass}}, \mathtt{s}'' \cdot \ell') = \mathtt{current}(\mathbf{c}', \Theta_{\mathsf{ass}}, \mathtt{s}'') + 1$$

whenever ℓ' is an input. Thus we have $\{\mathsf{ini}' + 1, ..., \mathsf{ini}' + \mathsf{num}(\mathtt{s}_0)\}$ is a contiguous set with minimun $\mathsf{ini}' + 1$ and maximum $\mathsf{ini}' + \mathsf{num}(\mathtt{s}_0)$.

(b) Up to $\mathtt{s}_0{\cdot}\ell$, by cond 5,

$$\mathsf{ini} \leq \mathtt{v}(\ell) < \mathsf{num}(\mathtt{I}(\mathtt{s}_0{\cdot}\ell)) + \mathsf{ini} = \mathsf{num}(\mathtt{I}(\mathtt{s}_0)) + \mathsf{ini} = \mathsf{num}(\mathtt{s}_0) + \mathsf{ini}$$

which implies $\mathtt{v}(\ell) \in \mathtt{current}(\mathbf{log}, \Theta_{\mathsf{ass}}, \mathtt{s}_0)$ so that Θ_{ass} accepts $\mathtt{s}_0{\cdot}\ell$. Note that, for any $\mathtt{s}$, $\mathsf{num}(\mathtt{I}(\mathtt{s}_0{\cdot}\ell)) = \mathsf{num}(\mathtt{I}(\mathtt{s}_0))$ whenever ℓ is an output.

(c) Assume for any $\mathtt{s}' \in \mathtt{prefix}(\mathtt{s})$, $\mathtt{s}' = \mathtt{s}_f{\cdot}\ell$. Assume Θ_{ass} accepts $\mathtt{s}_f$ and let $\{\ell \mid \ell \in \mathtt{O}(\mathtt{s}_f)\} = \mathtt{out}(\mathtt{s}_f)$ be the set of output *values* happening in $\mathtt{s}_f$. So, up to $\mathtt{s}_f$, we have

$$\mathtt{current}(\mathbf{log}, \Theta_{\mathsf{ass}}, \mathtt{s}_f) = \{\mathsf{ini}' + 1, ..., \mathsf{ini}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_f))\} \setminus \mathtt{out}(\mathtt{s}_f)$$

where $\{\mathsf{ini}' + 1, ..., \mathsf{ini}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_f))\}$ is a contiguous series.

Firstly, it needs to prove that $\{\mathsf{ini}' + 1, ..., \mathsf{ini}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_f))\}$ is a contiguous series. Assume $\mathtt{s}_f = \mathtt{s}_0 \cdot \mathtt{s}'_f$ where $\mathtt{s}_0$ only contains inputs, then it is straightforward that $\{\mathsf{ini}'+1, ..., \mathsf{ini}'+\mathsf{num}(\mathtt{I}(\mathtt{s}_0))\}$ is a contiguous series simply from (a). The following claims that, if a set is a contiguous series, then after adding new elements to it because of inputs, the resulting set is still a contiguous series.

Claim. For $\mathtt{s}_0{\cdot}\mathtt{s}'_1 = \mathtt{s}', \mathtt{s}' \in \mathtt{prefix}(\mathtt{s})$ where $\mathtt{s}_0$ only contains inputs, and let $\mathtt{out}(\mathtt{s}'_1)$ be the set of output values of $\mathtt{s}'_1$. If $\mathtt{current}(\mathbf{log}, \Theta_{\mathsf{ass}}, \mathtt{s}_0) = \{\mathsf{ini}' + 1, ..., \mathsf{ini}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_0))\}$, then

$$\begin{aligned}
&\mathtt{current}(\mathbf{log}, \Theta_{\mathsf{ass}}, \mathtt{s}_0{\cdot}\mathtt{s}'_1) \\
&= \{\mathsf{ini}' + 1, ..., \mathsf{ini}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_0))\} \cup \\
&\quad \{\mathsf{ini}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_0)) + 1, ..., \mathsf{ini}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_0)) + \mathsf{num}(\mathtt{I}(\mathtt{s}'_1))\} \setminus \mathtt{out}(\mathtt{s}'_1) \\
&= \{\mathsf{ini}' + 1, ..., \mathsf{ini}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_0)) + \mathsf{num}(\mathtt{I}(\mathtt{s}'_1))\} \setminus \mathtt{out}(\mathtt{s}'_1) \\
&= \{\mathsf{ini}' + 1, ..., \mathsf{ini}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_0{\cdot}\mathtt{s}'_1))\} \setminus \mathtt{out}(\mathtt{s}'_1)
\end{aligned}$$

and the set $\{\mathsf{ini}' + 1, ..., \mathsf{ini}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_0{\cdot}\mathtt{s}'_1))\}$ is a contiguous series.

This claim can be proved by the following reasoning: according to Θ_{ass}, once an input happens, it updates $\mathbf{c}' := \mathbf{c}' + 1$ and $\mathbf{log} := \mathbf{log} \cup \{\mathbf{c}'\}$. The new ele-

ments added by the inputs in $\mathtt{s}'_1$ are in the range from $\mathtt{current}(\mathbf{c}', \Theta_{\mathsf{ass}}, \mathtt{s}_0{\cdot}\ell)$, where ℓ is the first input of $\mathtt{s}'_1$, to $\mathtt{current}(\mathbf{c}', \Theta_{\mathsf{ass}}, \mathtt{s}_0 \cdot \mathtt{s}'_1)$ where

$$\begin{aligned} & \mathtt{current}(\mathbf{c}', \Theta_{\mathsf{ass}}, \mathtt{s}_0{\cdot}\ell) \\ = \;& \mathtt{current}(\mathbf{c}', \Theta_{\mathsf{ass}}, \mathtt{s}_0) + 1 \\ = \;& \mathsf{ini}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_0)) + 1 \end{aligned}$$

and

$$\begin{aligned} & \mathtt{current}(\mathbf{c}', \Theta_{\mathsf{ass}}, \mathtt{s}_0 \cdot \mathtt{s}'_1) \\ = \;& \mathtt{current}(\mathbf{c}', \Theta_{\mathsf{ass}}, \mathtt{s}_0) + \mathsf{num}(\mathtt{I}(\mathtt{s}'_1)) \\ = \;& \mathsf{ini}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_0)) + \mathsf{num}(\mathtt{I}(\mathtt{s}'_1)) \end{aligned}$$

Because $\mathsf{num}(\mathtt{I}(\mathtt{s}_0)) + \mathsf{num}(\mathtt{I}(\mathtt{s}'_1)) = \mathsf{num}(\mathtt{I}(\mathtt{s}_0 \cdot \mathtt{s}'_1))$, the set of new elements added to $\mathtt{current}(\mathbf{log}, \Theta_{\mathsf{ass}}, \mathtt{s}_0)$ is

$$\{\mathsf{ini}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_0)) + 1, ..., \mathsf{ini}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_0 \cdot \mathtt{s}'_1))\}$$

which is a contiguous series set by the same reasoning of (a). Because the minimum element of set $\{\mathsf{ini}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_0)) + 1, ..., \mathsf{ini}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_0 \cdot \mathtt{s}'_1))\}$, which is $\mathsf{ini}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_0)) + 1$, is one more bigger than the maximum element of set $\{\mathsf{ini}', ..., \mathsf{ini}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_0))\}$, which is $\mathsf{ini}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_0))$, the union of these two contiguous sets is still a contiguous series set. Moreover,

$$\mathtt{current}(\mathbf{log}, \Theta_{\mathsf{ass}}, \mathtt{s}_0 \cdot \mathtt{s}'_1) = \{\mathsf{ini}' + 1, ..., \mathsf{ini}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_0 \cdot \mathtt{s}'_1))\} \setminus \mathtt{out}(\mathtt{s}'_1)$$

simply according to the updates $\mathbf{c}' := \mathbf{c}' + 1$ and $\mathbf{log} := \mathbf{log} \cup \{\mathbf{c}'\}$ for every input, and the update $\mathbf{log} := \mathbf{log} \setminus \mathtt{v}(\ell)$ for every output ℓ.

Continue to prove that $\mathtt{s}_f{\cdot}\ell$ is anyway accepted by Θ_{ass}. When ℓ is an input, Θ_{ass} accepts $\mathtt{s}_f{\cdot}\ell$ because Θ_{ass} accepts any input. When ℓ is an output, then by cond 5, $\mathsf{ini} \leq \mathtt{v}(\ell) < \mathsf{ini} + \mathsf{num}(\mathtt{I}(\mathtt{s}_f{\cdot}\ell)) = \mathsf{ini} + \mathsf{num}(\mathtt{I}(\mathtt{s}_f))$, we know that $\mathtt{v}(\ell) \in \{\mathsf{ini}' + 1, ..., \mathsf{ini}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_f))\}$ because it is a contiguous series set with $\mathsf{ini}' = \mathsf{ini} - 1$.

Further, by cond 4 we know $\mathtt{v}(\ell)$ is different from any value of actions in

$\mathtt{out}(\mathtt{s}_f)$. They together imply $\mathtt{v}(\ell)$ is in

$$\mathtt{current}(\mathbf{log}, \Theta_{\mathsf{ass}}, \mathtt{s}_f) = \{\mathsf{ini}' + 1, ..., \mathsf{ini}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_f))\} \setminus \mathtt{out}(\mathtt{s}_f)$$

Therefore, by induction, every output in $\mathtt{s}$ is accepted by Θ_{ass}, which means every $\mathtt{s}$ satisfies all conditions in Lemma 8.5.13, $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{ass}})$.

(For $\Leftarrow$): Assume $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{ass}})$ is given. The reasonings are as follows:

1. $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{ass}})$ means that the sessions declared in $\mathtt{s}$ has been validly established (as $s_i, i \in I$), which implies $\mathtt{s}$ satisfies cond 1.

2. By the definition of Θ_{ass}, if $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{ass}})$, then the inputs should always happen before the outputs in the same session, thus $\mathtt{s}$ satisfies cond 2, and cond 3.

3. $\mathbf{log}$ is updated while an input happens by updating $\mathbf{log} := \mathbf{log} \cup \{\mathbf{c}'\}$; therefore, for any v_1 and v_2 in $\mathbf{log}$, v_1 is always different from v_2 simply because whenever an input happens, $\mathbf{c}'$ updates to a different value (i.e. $\mathbf{c}' := \mathbf{c}' + 1$) and this new value is added to set $\mathbf{log}$ (i.e. $\mathbf{log} := \mathbf{log} \cup \{\mathbf{c}'\}$). Since every outputted value v in $\mathtt{s}$ should be selected from set $\mathbf{log}$ through verifying the predicate $v \in \mathbf{log}$, and, when outputting v, $\mathbf{log}$ is updated by $\mathbf{log} := \mathbf{log} \setminus \{v\}$, they together ensure that for any outputs ℓ and ℓ' in $\mathtt{s}$, $\mathtt{v}(\ell) \neq \mathtt{v}(\ell')$. Thus $\mathtt{s}$ satisfies condition 3.

4. By the definition of Θ_{ass}, the initial value of $\mathbf{c}'$ is ini' and $\mathbf{c}'$ is updated by increasing one when an input happens. Therefore, for every $\mathtt{s}' \in \mathtt{prefix}(\mathtt{s})$, again, $\mathtt{current}(\mathbf{c}', \Theta_{\mathsf{ass}}, \mathtt{s}')$ defined before is used to represent the lastest value of $\mathbf{c}'$ when the actions in $\mathtt{s}'$ have happened. Then we know

   $$\mathtt{current}(\mathbf{c}', \Theta_{\mathsf{ass}}, \mathtt{s}') = \mathsf{num}(\mathtt{I}(\mathtt{s}')) + \mathsf{ini}'$$

   because the lastest value of $\mathbf{c}'$ is the summation of the number of inputs happening in $\mathtt{s}'$ plus the starting value ini'. Since for every outputted value v in $\mathtt{s}' \in \mathtt{prefix}(\mathtt{s})$, there exists an input belonging to the same session of v ahead of it, and it is selected from set $\mathbf{log}$ by verifying through predicate $v \in \mathbf{log}$, v is always in the range from $\mathsf{ini}' + 1$, the minimum element in $\mathbf{log}$, to $\mathsf{num}(\mathtt{I}(\mathtt{s}')) + \mathsf{ini}'$,

the maximum element in **log**. Therefore we have

$$\mathsf{ini}' + 1 \leq v \leq \mathtt{current}(\mathbf{c}', \Theta_{\mathsf{ass}}, \mathtt{s}') = \mathsf{num}(\mathtt{I}(\mathtt{s}')) + \mathsf{ini}',$$

which implies

$$\mathsf{ini} \leq v \leq \mathsf{num}(\mathtt{I}(\mathtt{s}')) + \mathsf{ini}' < \mathsf{num}(\mathtt{I}(\mathtt{s}')) + \mathsf{ini}$$

due to $\mathsf{ini}' = \mathsf{ini} - 1$. So that $\mathtt{s}$ satisfies cond 5.

5. Therefore, $\forall \mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{ass}})$, $\mathtt{s}$ satisfies all conditions defined in Lemma 8.5.13.■

Proposition 8.5.19. *$P \models_{\mathtt{sync}} \Theta_{\mathsf{ass}}$ then $P \models_{\mathtt{async}} \Theta_{\mathsf{ass}}$.*

Proof. It can be proved simply by proving that Θ_{ass} is commutative (so that it is asynchronous verifiable), or proved by the following reasoning. Let ℓ^{in} represent an input action, and ℓ^{out} represent an output action. For every $\mathtt{s} \in \mathsf{Obs}_s(P)$, in which $P \models_{\mathtt{sync}} \Theta_{\mathsf{ass}}$, according to Θ_{ass}, it is an input-output alternating sequence with the following shape

$$\mathtt{s} = \ell_1^{in} \cdot \ell_1^{out} \cdot \ell_2^{in} \cdot \ell_2^{out} .....$$

which will end up with ℓ_k^{in} or $\ell_k^{in} \cdot \ell_k^{out}$ for some k, and satisfies all conditions listed in Lemma 8.5.13. Since the permutations only move inputs ahead of outputs, no condition is affected by the permutations. Thus $\forall \mathtt{s} \curvearrowright \mathtt{s}', \mathtt{s}' \in \mathtt{trace}(\Theta_{\mathsf{ass}})$ i.e. $\mathtt{s}' \in \mathsf{Obs}_a(P), P \models_{\mathtt{async}} \Theta_{\mathsf{ass}}$.■

Lemma C.1.11. Let ℓ_i^{out} be the ith output action and ℓ_i^{in} be the ith input action. $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{sync}})$ if and only if

1. $\ell_1^{out} = \mathsf{ini}$, and
2. for any $\ell_i^{out} \in \mathtt{s}$, there exist ℓ_i^{in} positioning before ℓ_i^{out}, and
3. for any $\ell_i^{out}, \ell_{i-1}^{out} \in \mathtt{s}$, $\mathtt{v}(\ell_i^{out}) = \mathtt{v}(\ell_{i-1}^{out}) + 1$.

Proof. For ($\Rightarrow$), according to Θ_{sync} and Lemma C.1.2, for any $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{sync}})$, let $\mathtt{s} = \mathtt{s}_0 \cdot \ell_{i-1}^{out} \cdot \mathtt{s}_0' \cdot \ell_i^{out} \cdot \mathtt{s}_1$, where $\mathtt{s}_0'$ only contains input actions because ℓ_i^{out} is the next output action after ℓ_{i-1}^{out}. The three conditions are proved as followings.

C. APPENDIX FOR SPECIFYING STATEFUL ASYNCHRONOUS PROPERTIES FOR DISTRIBUTED PROGRAMS

1. For the value of the first output according to Θ_{sync}, let $\mathtt{s}_0$ contains only inputs and $i = 2$, then we have $\mathtt{v}(\ell_1^{out}) = \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}_0) = \mathsf{num}(\mathtt{O}(\mathtt{s}_0)) + \mathsf{ini} = \mathsf{ini}$.

2. According to Θ_{sync}, whenever $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{sync}})$, there always exists ℓ_{i-1}^{in} in $\mathtt{s}_0$ and ℓ_i^{in} in $\mathtt{s}_0'$.

3. Moreover we have

$$\mathtt{v}(\ell_{i-1}^{out}) = \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}_0) = \mathsf{num}(\mathtt{O}(\mathtt{s}_0)) + \mathsf{ini}$$

and

$$\mathtt{v}(\ell_i^{out}) = \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}_0 \cdot \ell_{i-1}^{out} \cdot \mathtt{s}_0') = \mathsf{num}(\mathtt{O}(\mathtt{s}_0 \cdot \ell_{i-1}^{out} \cdot \mathtt{s}_0')) + \mathsf{ini}$$

where

$$\mathsf{num}(\mathtt{O}(\mathtt{s}_0 \cdot \ell_{i-1}^{out} \cdot \mathtt{s}_0')) + \mathsf{ini} = \mathsf{num}(\mathtt{O}(\mathtt{s}_0)) + 1 + \mathsf{ini}$$

therefore we have $\mathtt{v}(\ell_i^{out}) = \mathtt{v}(\ell_{i-1}^{out}) + 1$ for any ℓ_i^{out} and ℓ_{i-1}^{out} in such $\mathtt{s}$.

For ($\Leftarrow$), the prove is done by induction. Assume $\mathtt{s}_0'' \in \mathtt{prefix}(\mathtt{s})$, $\mathtt{s}_0'' \in \mathtt{trace}(\Theta_{\mathsf{sync}})$. We can always find such $\mathtt{s}_0''$ by letting $\mathtt{s}_0''$ be a trace which only contains input actions. Consider $\mathtt{s}_0'' \cdot \ell_1^{out}$, where ℓ_1^{out} is the first output so that $\mathtt{s}_0''$ contains only inputs,

$$\mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}_0'') \quad = \quad \mathsf{num}(\mathtt{O}(\mathtt{s}_0'')) + \mathsf{ini} = \mathsf{ini} = \mathtt{v}(\ell_1^{out})$$

which means that $\mathtt{s}_0 \cdot \ell_1^{out}$ is valid to Θ_{sync} because the outputted value is equal to the current value of state $\mathbf{c}$.

Let $\mathtt{s}_0'' = \mathtt{s}_0 \cdot \ell_{i-1}^{out}$ where $\mathtt{s}_0$ may contain some inputs and outputs or only inputs. Without loss of generality, assume $\mathtt{s}_0''$ is valid according to Θ_{sync}. It means

$$\mathtt{v}(\ell_{i-1}^{out}) = \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}_0) = \mathsf{num}(\mathtt{O}(\mathtt{s}_0)) + \mathsf{ini}$$

Consider trace $\mathtt{s}_0 \cdot \ell_{i-1}^{out} \cdot \mathtt{s}_0' \cdot \ell_i^{out}$, where ℓ_i^{out} is the next output of ℓ_{i-1}^{out} and $\mathtt{s}_0'$ only contains inputs. Because Θ_{sync} always approves input actions, it implies that $\mathtt{s}_0 \cdot \ell_{i-1}^{out} \cdot \mathtt{s}_0'$ is valid according to Θ_{sync}. Because for every $\mathtt{v}(\ell_i^{out}), \mathtt{v}(\ell_{i-1}^{out}) \in \mathtt{s}$, we always have

$\mathtt{v}(\ell_i^{out}) = \mathtt{v}(\ell_{i-1}^{out}) + 1$, therefore, we have the following equations:

$$\begin{aligned}
\mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}_0 \cdot \ell_{i-1}^{out} \cdot \mathtt{s}_0') &= \mathsf{num}(\mathtt{O}(\mathtt{s}_0 \cdot \ell_{i-1}^{out} \cdot \mathtt{s}_0')) + \mathsf{ini} \\
&= \mathsf{num}(\mathtt{O}(\mathtt{s}_0)) + 1 + \mathsf{ini} \\
&= \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}_0) + 1 \\
&= \mathtt{v}(\ell_{i-1}^{out}) + 1 = \mathtt{v}(\ell_i^{out})
\end{aligned}$$

which means the outputted value is equal to the current value of state $\mathbf{c}$, so that $\mathtt{s}_0 \cdot \ell_{i-1}^{out} \cdot \mathtt{s}_0' \cdot \ell_i^{out}$ is valid according to Θ_{sync}. By Induction, for any $\mathtt{s}$ satisfying these three conditions, we know $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{sync}})$.■

Proposition 8.5.20. Assume $\mathtt{s}$ is an input-output alternating sequence. If $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{ass}})$, then $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{sync}})$.

Proof. It is proved by induction. $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{ass}})$ if and only if $\mathtt{s}$ satisfies all conditions listed in Lemma 8.5.13 gives the following reasonings:

1. For the first output action ℓ_1^{out}, by cond 5, $\mathsf{ini} \leq \mathtt{v}(\ell_1^{out}) < \mathsf{num}(\mathtt{I}(\ell_1^{in} \cdot \ell_1^{out})) + \mathsf{ini}$. Due to $\mathsf{ini}' = \mathsf{ini} - 1$ and according to Θ_{ass}, we have

   $$\mathtt{v}(\ell_1^{out}) \in \{\mathsf{ini}' + 1\} = \{\mathsf{ini}\}$$

   which means $\mathtt{v}(\ell_1^{out}) = \mathsf{ini}$.

2. For the second output action ℓ_2^{out}, by cond 5, $\mathsf{ini} \leq \mathtt{v}(\ell_2^{out}) < \mathsf{num}(\mathtt{I}(\ell_1^{in} \cdot \ell_1^{out} \cdot \ell_2^{in} \cdot \ell_2^{out})) + \mathsf{ini}$. Due to $\mathsf{ini}' = \mathsf{ini} - 1$ and according to Θ_{ass}, we have

   $$\begin{aligned}
   \mathtt{v}(\ell_2^{out}) \quad \in \quad & \{\mathsf{ini}' + 1, \mathsf{ini}' + \mathsf{num}(\mathtt{I}(\ell_1^{in} \cdot \ell_1^{out} \cdot \ell_2^{in} \cdot \ell_2^{out}))\} \setminus \{\mathsf{ini}' + 1\} \\
   & \{\mathsf{ini}' + 1, \mathsf{ini}' + 2\} \setminus \{\mathsf{ini}' + 1\}
   \end{aligned}$$

   which means $\mathtt{v}(\ell_2^{out}) = \mathsf{ini}' + 2 = \mathsf{ini} + 1$.

3. Let $\mathtt{s}_0 = \ell_1^{in} \cdot \ell_1^{out} ... \ell_{i-1}^{in} \cdot \ell_{i-1}^{out}$. Assume for the ith output action ℓ_i^{out} we have

   $$\mathtt{v}(\ell_i^{out}) \quad \in \quad \{\mathsf{ini}' + 1, ... \mathsf{ini}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_0 \cdot \ell_i^{in} \cdot \ell_i^{out}))\} \setminus \{\mathsf{ini}' + 1, ..., \mathsf{ini}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_0))\}$$

so that

$$\mathtt{v}(\ell_i^{out}) \quad \in \quad \{\mathsf{ini}' + 1, ..., \mathsf{ini}' + (i-1), \mathsf{ini}' + i\} \setminus \{\mathsf{ini}' + 1, ..., \mathsf{ini}' + (i-1)\}$$

which means $\mathtt{v}(\ell_i^{out}) = \mathsf{ini}' + i = \mathsf{ini} + (i-1)$.

Then for the $(i+1)$th output action ℓ_{i+1}^{out}, by cond 5, $\mathsf{ini} \leq \mathtt{v}(\ell_{i+1}^{out}) < \mathsf{num}(\mathtt{I}(\mathtt{s}_0 \cdot \ell_i^{in} \cdot \ell_i^{out} \cdot \ell_{i+1}^{in} \cdot \ell_{i+1}^{out}) + \mathsf{ini}$. Due to $\mathsf{ini}' = \mathsf{ini} - 1$ and according to Θ_{ass}, we have

$$\begin{array}{rcl} \mathtt{v}(\ell_{i+1}^{out}) & \in & \{\mathsf{ini}' + 1, ...\mathsf{ini}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_0 \cdot \ell_i^{in} \cdot \ell_i^{out} \cdot \ell_{i+1}^{in} \cdot \ell_{i+1}^{out}))\} \\ & & \setminus \{\mathsf{ini}' + 1, ..., \mathsf{ini}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_0 \cdot \ell_i^{in} \cdot \ell_i^{out}))\} \end{array}$$

so that

$$\mathtt{v}(\ell_{i+1}^{out}) \quad \in \quad \{\mathsf{ini}' + 1, ..., \mathsf{ini}' + i, \mathsf{ini}' + i + 1\} \setminus \{\mathsf{ini}' + 1, ..., \mathsf{ini}' + i\}$$

therefore we have $\mathtt{v}(\ell_{i+1}^{out}) = \mathsf{ini}' + (i+1) = \mathsf{ini} + i$.

Finally we have $\mathtt{v}(\ell_{i+1}^{out}) = \mathtt{v}(\ell_i^{out}) + 1$. Therefore, by Lemma C.1.11, $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{sync}})$. ■

Note that, although Proposition 8.5.14 says that the set of traces satisfying conditions listed in Lemma 8.5.13 is equal to $\mathtt{trace}(\Theta_{\mathsf{ass}})$, and Proposition 8.5.20 says that G_{assign} is an asynchronously verifiable specification realising synchronous behaviours w.r.t Θ_{sync}, $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{ass}})$ does not imply $\mathtt{s} \in \mathsf{Obs}_a(P)$ such that $P \models_{\mathtt{sync}} \Theta_{\mathsf{sync}}$. It is simply because $\mathtt{trace}(\Theta_{\mathsf{ass}})$ is bigger than $\mathsf{Obs}_a(P)$ in which $P \models_{\mathtt{sync}} \Theta_{\mathsf{sync}}$. The following example illustrates this point.

Example C.1.12. Consider a sequence of actions below:

$$s_1[C,S]?\mathrm{req}(\varepsilon) \cdot s_2[C,S]?\mathrm{req}(\varepsilon) \cdot s_1[S,C]!\mathrm{ans}(v)$$

if it is a trace in $\mathsf{Obs}_a(P)$ such that $P \models_{\mathtt{sync}} \Theta_{\mathsf{sync}}$, then v can only be 1; however if it is a trace in $\mathtt{trace}(\Theta_{\mathsf{ass}})$, v can be either 1 or 2.

C.2 Proofs for SPs' Commutativity

Here we illustrate the main obligations in the SPs introduced in examples (see § 8.1 and § 8.1.4). Note that, only the state(s) updated at every input/output action appear(s)

in the obligations.

Proof (G_{sync} is not asynchronous). Assume $\xi^{\mathtt{i}}_{\mathsf{sync}}$ (for input) and $\xi^{\mathtt{o}}_{\mathsf{sync}}$ (for output) are the notations for the obligations in G_{sync}.

$$\begin{aligned} \xi^{\mathtt{i}}_{\mathsf{sync}} &\stackrel{\text{def}}{=} B?\mathrm{req}(\varepsilon)\langle \mathsf{true}; \varepsilon\rangle \\ \xi^{\mathtt{o}}_{\mathsf{sync}} &\stackrel{\text{def}}{=} B!\mathrm{ans}(x : \mathsf{int})\langle x = \mathbf{c}; \mathbf{c} := \mathbf{c} + 1\rangle \end{aligned}$$

$$\begin{aligned} \mathtt{pred}(\xi^{\mathtt{i}}_{\mathsf{sync}}) &\stackrel{\text{def}}{=} \lambda\varepsilon, \mathbf{c}.(\mathsf{true}) \\ \mathtt{pred}(\xi^{\mathtt{o}}_{\mathsf{sync}}) &\stackrel{\text{def}}{=} \lambda x, \mathbf{c}.(x = \mathbf{c}) \\ \mathtt{upd}(\xi^{\mathtt{i}}_{\mathsf{sync}}) &\stackrel{\text{def}}{=} \lambda\varepsilon, \mathbf{c}.\langle\varepsilon\rangle \\ \mathtt{upd}(\xi^{\mathtt{o}}_{\mathsf{sync}}) &\stackrel{\text{def}}{=} \lambda x, \mathbf{c}.\langle \mathbf{c} + 1\rangle \end{aligned}$$

Then for $\xi^{\mathtt{o}}_{\mathsf{sync}}, \xi^{\mathtt{o}}_{\mathsf{sync}}$, we start from $\mathbf{c}$'s current value is $w = 1$:

$$\mathbf{c} = w = 1.$$

$$\mathtt{pred}(\xi^{\mathtt{o}}_{\mathsf{sync}})(1, 1) = \mathsf{true}$$

and

$$\mathtt{pred}(\xi^{\mathtt{o}}_{\mathsf{sync}})(2, \mathtt{upd}(\xi^{\mathtt{o}}_{\mathsf{sync}})(1, 1)) = \mathtt{pred}(\xi^{\mathtt{o}}_{\mathsf{sync}})(2, 2) = \mathsf{true}.$$

Then we have

$$\mathtt{pred}(\xi^{\mathtt{o}}_{\mathsf{sync}})(2, 1) = \mathsf{false}.$$

Thus condition 1. in Definition 8.5.29 does not hold. Similarly, $\xi^{\mathtt{i}}_{\mathsf{sync}}$ and $\xi^{\mathtt{o}}_{\mathsf{sync}}$ are not commutative. ■

Proof (G_{assign} is asynchronous). Assume $\xi^{\mathtt{i}}_{\mathsf{ass}}$ and $\xi^{\mathtt{o}}_{\mathsf{ass}}$ are the notations for obligations in G_{assign}.

$$\begin{aligned} \xi^{\mathtt{i}}_{\mathsf{ass}} &\stackrel{\text{def}}{=} B?\mathrm{req}(\varepsilon)\langle \mathsf{true}; \mathbf{t} := \mathbf{t} + 1 \;\; \mathbf{c} := \mathbf{c} \cup \{\mathbf{t}\}\rangle \\ \xi^{\mathtt{o}}_{\mathsf{ass}} &\stackrel{\text{def}}{=} B!\mathrm{ans}(x : \mathsf{int})\langle x \in \mathbf{c} \;;\; \mathbf{c} := \mathbf{c} \setminus \{x\}\rangle \end{aligned}$$

$$\begin{aligned} \mathtt{pred}(\xi^{\mathtt{i}}_{\mathsf{ass}}) &\stackrel{\text{def}}{=} \lambda\varepsilon, \mathbf{t}, \mathbf{c}.(\mathsf{true}) \\ \mathtt{pred}(\xi^{\mathtt{o}}_{\mathsf{ass}}) &\stackrel{\text{def}}{=} \lambda x, \mathbf{t}, \mathbf{c}.(x \in \mathbf{c}) \\ \mathtt{upd}(\xi^{\mathtt{i}}_{\mathsf{ass}}) &\stackrel{\text{def}}{=} \lambda\varepsilon, \mathbf{t}, \mathbf{c}.\langle \mathbf{t} + 1 \;\; \mathbf{c} \cup \{\mathbf{t}\}\rangle \\ \mathtt{upd}(\xi^{\mathtt{o}}_{\mathsf{ass}}) &\stackrel{\text{def}}{=} \lambda x, \mathbf{t}, \mathbf{c}.\langle \mathbf{c} \setminus \{x\}\rangle \end{aligned}$$

As for $\xi^{\mathsf{o}}_{\mathsf{ass}}, \xi^{\mathsf{o}}_{\mathsf{ass}}$, let $\mathbf{c} = w$, where w is a set. Assume

$$\mathtt{pred}(\xi^{\mathsf{o}}_{\mathsf{ass}})(v, w) = \mathsf{true}$$

and

$$\mathtt{pred}(\xi^{\mathsf{o}}_{\mathsf{ass}})(v', \mathtt{upd}(\xi^{\mathsf{o}}_{\mathsf{ass}})(v, w)) = \mathtt{pred}(\xi^{\mathsf{o}}_{\mathsf{ass}})(v', w \setminus \{v\}) = \mathsf{true}.$$

Then, $v' \neq v$, we have

$$\mathtt{pred}(\xi^{\mathsf{o}}_{\mathsf{ass}})(v', w) = \mathsf{true}$$

and

$$\mathtt{pred}(\xi^{\mathsf{o}}_{\mathsf{ass}})(v, \mathtt{upd}(\xi^{\mathsf{o}}_{\mathsf{ass}})(v', w)) = \mathtt{pred}(\xi^{\mathsf{o}}_{\mathsf{ass}})(v, w \setminus \{v'\}) = \mathsf{true}.$$

Moreover,

$$\mathtt{upd}(\xi^{\mathsf{o}}_{\mathsf{ass}})(v', \mathtt{upd}(\xi^{\mathsf{o}}_{\mathsf{ass}})(v, w)) = \mathtt{upd}(\xi^{\mathsf{o}}_{\mathsf{ass}})(v', w \setminus \{v\}) = w \setminus \{v\} \setminus \{v'\},$$

which is equal to

$$\mathtt{upd}(\xi^{\mathsf{o}}_{\mathsf{ass}})(v, \mathtt{upd}(\xi^{\mathsf{o}}_{\mathsf{ass}})(v', w)) = \mathtt{upd}(\xi^{\mathsf{o}}_{\mathsf{ass}})(v, w \setminus \{v'\})) = w \setminus \{v'\} \setminus \{v\}.$$

So that conditions 1. and 2. in Definition 8.5.29 both hold.

We can similarly check $\xi^{\mathsf{i}}_{\mathsf{ass}}, \xi^{\mathsf{i}}_{\mathsf{ass}}$. Let $\mathbf{t} = w_1$ and $\mathbf{c} = w_2$. w_1 is the current value of state $\mathbf{t}$, and w_2 is the current set of state $\mathbf{c}$. Condition 1. in Definition 8.5.29 holds immediately since the predicate is always true in $\xi^{\mathsf{i}}_{\mathsf{ass}}$. Condition 2. also holds immediately since the interaction variable is always ε.

As for $\xi^{\mathsf{o}}_{\mathsf{ass}}, \xi^{\mathsf{i}}_{\mathsf{ass}}$, let $\mathbf{t} = w_1$ and $\mathbf{c} = w_2$. w_1 and w_2 are the current sets of states $\mathbf{t}$ and $\mathbf{c}$. Assume

$$\mathtt{pred}(\xi^{\mathsf{o}}_{\mathsf{ass}})(v, w_2) = \mathsf{true}$$

and

$$\mathtt{pred}(\xi^{\mathsf{i}}_{\mathsf{ass}})(\varepsilon, \mathtt{upd}(\xi^{\mathsf{o}}_{\mathsf{ass}})(v, w_2)) = \mathtt{pred}(\xi^{\mathsf{i}}_{\mathsf{ass}})(\varepsilon, w_2 \setminus \{v\}) = \mathsf{true}.$$

Then we have

$$\mathtt{pred}(\xi^{\mathsf{i}}_{\mathsf{ass}})(\varepsilon, w_1 \;\; w_2) = \mathsf{true}$$

and

$$\mathtt{pred}(\xi^{\mathsf{o}}_{\mathsf{ass}})(v, \mathtt{upd}(\xi^{\mathsf{i}}_{\mathsf{ass}})(\varepsilon, w_1 \;\; w_2)) = \mathtt{pred}(\xi^{\mathsf{o}}_{\mathsf{ass}})(v, w_2 \cup \{w_1 + 1\}) = \mathsf{true}.$$

Moreover,

$$\mathtt{upd}(\xi^{\mathsf{i}}_{\mathsf{ass}})(\varepsilon, \mathtt{upd}(\xi^{\mathsf{o}}_{\mathsf{ass}})(v, w_2)) = \mathtt{upd}(\xi^{\mathsf{i}}_{\mathsf{ass}})(\varepsilon, w_1 \;\; w_2 \setminus \{v\}) = w_1 + 1, w_2 \setminus \{v\} \cup \{w_1 + 1\},$$

which is equal to

$$\mathtt{upd}(\xi^{\mathsf{o}}_{\mathsf{ass}})(v, \mathtt{upd}(\xi^{\mathsf{i}}_{\mathsf{ass}})(\varepsilon, w_1 \;\; w_2)) = \mathtt{upd}(\xi^{\mathsf{o}}_{\mathsf{ass}})(v, w_1{+}1 \;\; w_2 \cup \{w_1{+}1\})) = w_1{+}1, w_2 \cup \{w_1{+}1\} \setminus \{v\}.$$

However, $\xi^{\mathsf{i}}_{\mathsf{ass}}, \xi^{\mathsf{o}}_{\mathsf{ass}}$ are not forward commutative. Assume

$$\mathtt{pred}(\xi^{\mathsf{i}}_{\mathsf{ass}})(\varepsilon, w_1 \;\; w_2) = \mathsf{true}$$

and

$$\mathtt{pred}(\xi^{\mathsf{o}}_{\mathsf{ass}})(v, \mathtt{upd}(\xi^{\mathsf{i}}_{\mathsf{ass}})(\varepsilon, w_1 \;\; w_2)) = \mathtt{pred}(\xi^{\mathsf{o}}_{\mathsf{ass}})(v, w_1 + 1 \;\; w_2 \cup \{w_1 + 1\}) = \mathsf{true}.$$

However $\mathtt{pred}(\xi^{\mathsf{o}}_{\mathsf{ass}})(v, w_2)$ needs not be equal to truth. So $\xi^{\mathsf{o}}_{\mathsf{ass}}, \xi^{\mathsf{i}}_{\mathsf{ass}}$ are forward commutative, but not commutative.

Then by Definition 8.5.31, G_{assign} is commutative, thus G_{assign} is asynchronous by Propositions 8.5.34 and 8.5.25. ■

Proof (G_{async} is asynchronous). Assume $\xi^{\mathsf{i}}_{\mathsf{async}}$ and $\xi^{\mathsf{o}}_{\mathsf{async}}$ are the notations for obligations in G_{async}.

$$\begin{aligned} \xi^{\mathsf{i}}_{\mathsf{async}} &\stackrel{\text{def}}{=} B?\mathrm{req}(\varepsilon)\langle \mathsf{true}; \varepsilon\rangle \\ \xi^{\mathsf{o}}_{\mathsf{async}} &\stackrel{\text{def}}{=} B!\mathrm{ans}(x : \mathsf{int})\langle x \notin \mathbf{c} \;;\; \mathbf{c} := \mathbf{c} \cup \{x\}\rangle \end{aligned}$$

$$\begin{aligned} \mathtt{pred}(\xi^{\mathsf{i}}_{\mathsf{async}}) &\stackrel{\text{def}}{=} \lambda\varepsilon, \mathbf{c}.(\mathsf{true}) \\ \mathtt{pred}(\xi^{\mathsf{o}}_{\mathsf{async}}) &\stackrel{\text{def}}{=} \lambda x, \mathbf{c}.(x \notin \mathbf{c}) \\ \mathtt{upd}(\xi^{\mathsf{i}}_{\mathsf{async}}) &\stackrel{\text{def}}{=} \lambda\varepsilon, \mathbf{c}.\langle\varepsilon\rangle \\ \mathtt{upd}(\xi^{\mathsf{o}}_{\mathsf{async}}) &\stackrel{\text{def}}{=} \lambda x, \mathbf{c}.\langle \mathbf{c} \cup \{x\}\rangle \end{aligned}$$

As for $\xi^{\mathsf{o}}_{\mathsf{async}}, \xi^{\mathsf{o}}_{\mathsf{async}}$, let $\mathbf{c} = w$, where w is a set. Assume

$$\mathtt{pred}(\xi^{\mathsf{o}}_{\mathsf{async}})(v, w) = \mathsf{true}$$

and

$$\mathtt{pred}(\xi^{\mathsf{o}}_{\mathsf{async}})(v', \mathtt{upd}(\xi^{\mathsf{o}}_{\mathsf{async}})(v, w)) = \mathtt{pred}(\xi^{\mathsf{o}}_{\mathsf{async}})(v', w \cup \{v\}) = \mathsf{true}.$$

Then, since

$$\mathtt{pred}(\xi^{\mathsf{o}}_{\mathsf{async}})(v', w \cup \{v\}) = \mathsf{true}$$

implies $v' \neq v$ and $v' \notin w$, we have

$$\mathtt{pred}(\xi^{\mathsf{o}}_{\mathsf{async}})(v', w) = \mathsf{true}$$

and

$$\mathtt{pred}(\xi^{\mathsf{o}}_{\mathsf{async}})(v, \mathtt{upd}(\xi^{\mathsf{o}}_{\mathsf{async}})(v', w)) = \mathtt{pred}(\xi^{\mathsf{o}}_{\mathsf{async}})(v, w \cup \{v'\}) = \mathsf{true}.$$

Moreover,

$$\mathtt{upd}(\xi^{\mathsf{o}}_{\mathsf{async}})(v', \mathtt{upd}(\xi^{\mathsf{o}}_{\mathsf{async}})(v, w)) = \mathtt{upd}(\xi^{\mathsf{o}}_{\mathsf{async}})(v', w \cup \{v\}) = w \cup \{v\} \cup \{v'\},$$

which is equal to

$$\mathtt{upd}(\xi^{\mathsf{o}}_{\mathsf{async}})(v, \mathtt{upd}(\xi^{\mathsf{o}}_{\mathsf{async}})(v', w)) = \mathtt{upd}(\xi^{\mathsf{o}}_{\mathsf{async}})(v, w \cup \{v'\})) = w \cup \{v'\} \cup \{v\}.$$

So that conditions 1. and 2. in Definition 8.5.29 both hold.
We can easily prove that $\xi^{\mathsf{i}}_{\mathsf{async}}, \xi^{\mathsf{i}}_{\mathsf{async}}$, and $\xi^{\mathsf{i}}_{\mathsf{async}}, \xi^{\mathsf{o}}_{\mathsf{async}}$, and $\xi^{\mathsf{o}}_{\mathsf{async}}, \xi^{\mathsf{i}}_{\mathsf{async}}$ are semi-commutativities. By Definition 8.5.31, G_{assign} is commutative, thus G_{assign} is asynchronous by Propositions 8.5.34 and 8.5.25. ■

References

[1] The Java Modeling Language (JML) homepage. http://www.jmlspecs.org/. 160

[2] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Trans. Program. Lang. Syst.*, 15(4):706–734, Sept. 1993. 41

[3] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The Spi calculus. In *4th ACM Conference on Computer and Communications Security*, pages 36–47. ACM Press, 1997. 42

[4] M. Afshar. SOA governance: Framework and best practices. Technical report, Oracle, 2007. SOA governance and practices. 39

[5] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. 28

[6] I. Aktug, M. Dam, and D. Gurov. Provably correct runtime monitoring. *J. Log. Algebr. Program.*, 78(5):304–339, 2009. 36, 37

[7] R. M. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous π-calculus. *Theoretical Computer Science*, 195(2):291–324, 1998. Concurrency Theory. 197

[8] `http://www.amazon.com`. 12

[9] D. Ancona, M. Barbieri, and V. Mascardi. Global types for dynamic checking of protocol conformance of multi-agent systems (Extended Abstract). In P. Massazza, editor, *13th Italian Conference on Theoretical Computer Science (ICTCS 2012)*, pages 39–43, 2012. 39

[10] D. Ancona, S. Drossopoulou, and V. Mascardi. Automatic generation of self-monitoring mass from multiparty global session types in Jason. In *Declarative Agent Languages and Technologies (DALT 2012). Workshop Notes*, pages 1–17, 2012. 38, 39

[11] J. Anderson. Computer security technology planning study. Technical report esd-tr-73-51. *U.S. Air Force Electronic Systems Division, Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC)*, Oct. 1972. 3, 36

[12] R. J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley Publishing, 2 edition, 2008. 11, 12

REFERENCES

[13] E. Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 10(1):110–135, 1975. 28

[14] J. Bacon. Expectations and reality in distributed systems. Keynote address at IASTED International Conference on Parallel and Distributed Computing and Networks, Feb. 2005. Presentation slides: see also the associated paper. 12

[15] J. Bacon, D. Evans, D. M. Eyers, M. Miglivacca, P. Pietzuch, and B. Shand. Big ideas paper: Enforcing end-to-end application security in the cloud. In *Proceedings ACM/IFIP/Usenix Middleware*, pages 293–312, 2010. 5

[16] H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. J. Pace, G. Rosu, O. Sokolsky, and N. Tillmann, editors. *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, volume 6418 of *Lecture Notes in Computer Science*. Springer, 2010. 196

[17] M. Y. Becker, C. Fournet, and A. D. Gordon. SecPAL: Design and semantics of a decentralized authorization language. *Journal of Computer Security*, 18(4):619–665, 2010. 1

[18] L. Bettini et al. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008. 158

[19] K. P. Birman. *Reliable Distributed Systems: Technologies, Web Services, and Applications*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005. 1

[20] L. Bocchi, T.-C. Chen, R. Demangeon, K. Honda, and N. Yoshida. Monitoring networks through multiparty session types. To appear in *FORTE/FMOOD*, `http://www.eecs.qmul.ac.uk/~tcchen/FORTE2013/DynamicMonitoring.pdf`, 2013. 8, 146

[21] L. Bocchi, R. Demangeon, and N. Yoshida. A multiparty multi-session logic, 2012. 197

[22] L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR*, LNCS, pages 162–176, 2010. 6, 7, 14, 26, 42, 92, 94, 95, 97, 98, 101, 102, 104, 139, 158, 165, 166, 168

[23] A. B. Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the 2nd international workshop on Software and performance*, WOSP '00, pages 195–203, New York, NY, USA, 2000. ACM. 2

[24] G. Boudol. Asynchrony and the pi-calculus. Technical Report 1702, INRIA, 1992. 13, 20

[25] M. Broy. Functional specification of time-sensitive communicating systems. *ACM Trans. Softw. Eng. Methodol.*, 2(1):1–46, Jan. 1993. 31

[26] L. Caires and H. T. Vieira. Conversation types. *Theor. Comput. Sci.*, 411(51-52):4399–4440, 2010. 195

[27] C. Caleiro, L. Viganò, and D. A. Basin. Relating strand spaces and distributed temporal logic for security protocol analysis. *Logic Journal of the IGPL*, 13(6):637–663, 2005. 29

[28] S. Capecchi, I. Castellani, and M. Dezani-Ciancaglini. Information flow safety in multiparty sessions. In *EXPRESS*, volume 64 of *EPTCS*, pages 16–30, 2011. 1

[29] S. Capecchi, I. Castellani, M. Dezani-Ciancaglini, and T. Rezk. Session types for access and information flow control. In *Proceedings of the 21st international conference on Concurrency theory*, CONCUR'10, pages 237–252, Berlin, Heidelberg, 2010. Springer-Verlag. 1

[30] M. Carbone, K. Honda, and N. Yoshida. A calculus of global interaction based on session types. *Electron. Notes Theor. Comput. Sci.*, 171(3):127–151, June 2007. 13, 14

[31] M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In *ESOP'07*, volume 4421 of *LNCS*, pages 2–17. Springer, 2007. 14

[32] M. Carbone, K. Honda, and N. Yoshida. Structured interactional exceptions in session types. In *CONCUR*, volume 5201 of *LNCS*, pages 402–417. Springer, 2008. 23

[33] L. Cardelli and A. D. Gordon. Mobile ambients. In *FoSSaCS*, pages 140–155, 1998. 14, 43

[34] G. Castagna, M. Dezani-Ciancaglini, and L. Padovani. On global types and multi-party sessions. In *FMOODS/FORTE*, pages 1–28, 2011. 57

[35] W3C WS-CDL. `http://www.w3.org/2002/ws/chor/`. 63, 110

[36] A. D. Chave, T. Ampe, M. Arrott, J. Graybeal, M. James, M. Meisinger, J. A. Orcutt, C. Peach, F. L. Vernon, and O. Schofield. Cyberinfrastructure for the US Ocean Observatories Initiative. In *Underwater Technology (UT), 2011 IEEE Symposium on and 2011 Workshop on Scientific Use of Submarine Cables and Related Technologies (SSC)*, pages 1–6, April 2011. 12

[37] F. Chen and G. Rosu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. *Electronic Notes in Theoretical Computer Science*, 89(2):108 – 127, 2003. RV '2003, Run-time Verification (Satellite Workshop of CAV '03). 37

[38] F. Chen and G. Rosu. MOP:An Efficient and Generic Runtime Verification Framework. In *OOPSLA*, pages 569–588, 2007. 37

[39] T.-C. Chen, L. Bocchi, P.-M. Deniélou, K. Honda, and N. Yoshida. Asynchronous distributed monitoring for multiparty session enforcement. In *TGC*, volume 7454 of *LNCS*, pages 25–45, 2011. 7, 39, 158, 203

[40] T.-C. Chen and K. Honda. Specifying stateful asynchronous properties for distributed programs. In *CONCUR*, pages 209–224, 2012. 7, 8, 157

[41] CIAD COI OV Governance Use Cases. `https://confluence.oceanobservatories.org/display/syseng/CIAD+COI+OV+Governance+Use+Cases`. 13

[42] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems, Concepts and Design*. Addison-Wesley, 2001. 1

[43] M. Dam, B. Jacobs, A. Lundblad, and F. Piessens. Security monitor inlining for multithreaded Java. In *ECOOP*, volume 5653 of *LNCS*, pages 546–569. Springer, 2009. 4, 6, 197

REFERENCES

[44] M. Dam, B. Jacobs, A. Lundblad, and F. Piessens. Provably correct inline monitoring for multithreaded java-like programs. *Journal of Computer Security*, 18(1):37–59, 2010. 37

[45] F. S. de Boer, J. N. Kok, C. Palamidessi, and J. J. M. M. Rutten. The failure of failures in a paradigm for asynchronous communication. In *CONCUR*, pages 111–126, 1991. 197

[46] R. De Nicola et al. From Flow Logic to static type systems for coordination languages. In *COORDINATION'08*, volume 5052 of *LNCS*, pages 100–116. Springer, 2008. 41

[47] R. De Nicola, G. Ferrari, and R. Pugliese. Klaim: a kernel language for agents interaction and mobility. *IEEE Tran. on Soft. Eng.*, 24:315–330, 1997. 41

[48] R. De Nicola, G. L. Ferrari, and R. Pugliese. Types as specifications of access policies. In *Secure Internet Programming*, pages 118–146, 1999. 41

[49] R. Demangeon and K. Honda. Nested protocols in session types. In *CONCUR*, volume 7454 of *LNCS*, pages 272–286, 2012. 22

[50] P.-M. Deniélou and N. Yoshida. Dynamic multirole session types. In *POPL*, pages 435–446, 2011. 102, 103, 195

[51] P.-M. Deniélou and N. Yoshida. Multiparty session types meet communicating automata. In *ESOP*, volume 7211 of *LNCS*, pages 194–213, 2012. 35

[52] G. Dennis, R. Seater, D. Rayside, and D. Jackson. Automating commutativity analysis at the design level. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '04, pages 165–174, New York, NY, USA, 2004. ACM. 187, 198

[53] Department of Defense. *Trusted Computer System Evaluation Criteria*. Dec. 1985. 1

[54] J. DeTreville. Binder, a logic-based security language. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, SP '02, pages 105–113, Washington, DC, USA, 2002. IEEE Computer Society. 1

[55] M. Dezani-Ciancaglini, S. Ghilezan, S. Jaksic, and J. Pantovic. Types for role-based access control of dynamic web data. In *WFLP*, pages 1–29, 2010. 41

[56] M. Dezani-Ciancaglini, S. Ghilezan, J. Pantovic, and D. Varacca. Security types for dynamic web data. *Theor. Comput. Sci.*, 402(2-3):156–171, 2008. 41

[57] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session types for object-oriented languages. In *ECOOP'06*, volume 4067 of *LNCS*, pages 328–352, 2006. 14

[58] E. Dijkstra. Hierarchical ordering of sequential processes. pages 29–52, 1971. 36

[59] U. Engberg and M. Nielsen. A calculus of communicating systems with label passing. Technical report, 1986. 14

[60] Y. Falcone. You should better enforce than verify. In *Runtime Verification*, Lecture Notes in Computer Science, pages 89–105. Springer, 2010. 196

[61] Y. Falcone, L. Mounier, J.-C. Fernandez, and J.-L. Richier. Runtime enforcement monitors: composition, synthesis, and enforcement abilities. *Formal Methods in System Design*, 38:223–262, 2011. 37

[62] D. Ferraiolo and R. Kuhn. Role-based access control. In *In 15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992. 1

[63] G. Ferrari, E. Moggi, and R. Pugliese. Guardians for ambient-based monitoring. In *F-WAN*, pages 141–202. Elsevier, 2002. 41

[64] R. W. Floyd. Assigning meaning to programs. In *Proceedings of the Symposium on Applied Maths*, volume 19, pages 19–32. AMS, 1967. 28

[65] R. Focardi and R. Gorrieri. Classification of security properties (part i: Information flow). In *Revised versions of lectures given during the IFIP WG 1.7 International School on Foundations of Security Analysis and Design on Foundations of Security Analysis and Design: Tutorial Lectures*, FOSAD '00, pages 331–396, London, UK, UK, 2001. Springer-Verlag. 1

[66] S. Ganeriwal, R. Kumar, and M. B. Srivastava. Timing-sync protocol for sensor networks. In *Proceedings of the 1st international conference on Embedded networked sensor systems*, SenSys '03, pages 138–149, New York, NY, USA, 2003. ACM. 34

[67] P. Gardner and S. Maffeis. Modelling dynamic web data. In G. Lausen and D. Suciu, editors, *Database Programming Languages*, volume 2921 of *Lecture Notes in Computer Science*, pages 130–146. Springer Berlin Heidelberg, 2004. 41

[68] P. Garralda, A. Compagnoni, and M. Dezani-Ciancaglini. Bass: boxed ambients with safe sessions. In *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '06, pages 61–72, New York, NY, USA, 2006. ACM. 43

[69] S. Gay, V. T. Vasconcelos, and A. Ravara. Session Types for Inter-Process Communication. TR 2003–133, Department of Computing, University of Glasgow, 2003. 13, 14

[70] Google. `http://www.google.com`. 12

[71] K. W. Hamlen and M. Jones. Aspect-oriented in-lined reference monitors. In *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, PLAS '08, pages 11–20, New York, NY, USA, 2008. ACM. 36

[72] P. Hansen. *Operating System Principles*. Prentice-Hall series in automatic computation. Prentice-Hall, 1973. 36

[73] K. Havelund and A. Goldberg. Verify your runs. In *Verified Software: Theories, Tools, Experiments*, pages 374–383. Springer, 2008. 35, 196

[74] B. Hayes. Cloud computing. *Commun. ACM*, 51(7):9–11, July 2008. 1

[75] M. Hennessy, J. Rathke, and N. Yoshida. safeDpi: a language for controlling mobile code. *Acta Inf.*, 42(4-5):227–290, 2005. 41

[76] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245, 1973. 14

[77] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct. 1969. 28

[78] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, Oct. 1974. 36

[79] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, Aug. 1978. 14, 19, 91

[80] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985. 14, 19, 91

[81] T. Hoare and H. Jifeng. *Unifying Theories of Programming.* Prentice-Hall International, 1998. 7, 184

[82] K. Honda. A theory of types for the π-calculus. Available at: www.dcs.qmul.ac.uk/˜kohei/logics, March 2001. 14

[83] K. Honda, A. Mukhamedov, G. Brown, T.-C. Chen, and N. Yoshida. Scribbling interactions with a formal foundation. In *ICDCIT*, volume 6536 of *LNCS*, pages 55–75, 2011. 7, 198

[84] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *ECOOP'91*, volume 512 of *LNCS*, pages 133–147, 1991. 13, 20, 197

[85] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998. 13

[86] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL'08*, pages 273–284. ACM, 2008. 4, 6, 13, 14, 23, 24, 57, 92, 94, 96, 101, 143, 158, 195

[87] K. Honda, N. Yoshida, R. Hu, P.-M. Deniélou, R. Neykova, R. Demangeon, and T. Chen. Scribble Development Team. 7

[88] R. Hu, D. Kouzapas, O. Pernet, N. Yoshida, and K. Honda. Type-safe eventful sessions in Java. In *ECOOP'10*, volume 6183 of *LNCS*, pages 329–353. Springer-Verlag, 2010. 183

[89] R. Hu, N. Yoshida, and K. Honda. Session-based distributed programming in Java. In *ECOOP'08*, volume 5142 of *LNCS*, pages 516–541. Springer, 2008. 14

[90] H. Jifeng, M. B. Josephs, and C. A. R. Hoare. A theory of synchrony and asynchrony. In *Programming Concepts and Methods*, IFIP, 1990. 197

[91] S.-M. Jun, D.-H. Yu, Y.-H. Kim, and S.-Y. Seong. A time synchronization method for NTP. In *Real-Time Computing Systems and Applications, 1999. RTCSA '99. Sixth International Conference on*, pages 466–473, 1999. 34

[92] M. Kajko-Mattsson, G. Lewis, and D. Smith. A framework for roles for development, evolution and maintenance of SOA-based systems. In *Systems Development in SOA Environments, 2007. SDSOA '07: ICSE Workshops 2007. International Workshop on*, page 7, may 2007. 39

[93] J.-L. Koning and P.-Y. Oudeyer. Formalization, implementation and validation of conversation policies using a protocol operational semantics. *Cognitive Systems Research*, 4(3):223 – 242, 2003. Cognitive Agents and Multiagent Interaction. 33

[94] K. Kontogiannis, G. A. Lewis, and D. B. Smith. A research agenda for service-oriented architecture. In *Proceedings of the 2nd international workshop on Systems development in SOA environments*, SDSOA '08, pages 1–6, New York, NY, USA, 2008. ACM. 39

[95] S. Kumar, M. J. Huber, P. R. Cohen, and D. R. McGee. Toward a formalism for conversation protocols using joint intention theory. *Computational Intelligence*, 18(2):174–228, 2002. 31, 32

[96] Y. Labrou, T. Finin, and Y. Peng. Agent communication languages: the current landscape. *Intelligent Systems and their Applications, IEEE*, 14(2):45–52, 1999. 33

[97] S. S. Lam and A. U. Shankar. Protocol verification via projections. *Software Engineering, IEEE Transactions on*, SE-10(4):325 –342, july 1984. 31

[98] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–564, July 1978. 34, 163, 197

[99] L. Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, 5(2):190–222, Apr. 1983. 29, 31

[100] L. Lamport. A simple approach to specifying concurrent systems. *Commun. ACM*, 32(1):32–45, 1989. 28

[101] L. Lamport. Verification and specifications of concurrent programs. In *REX School/Symposium*, pages 347–374, 1993. 28

[102] L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *J. ACM*, 32(1):52–78, Jan. 1985. 34

[103] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: theory and practice. *ACM Trans. Comput. Syst.*, 10(4):265–310, Nov. 1992. 3, 41

[104] B. W. Lampson. Dynamic protection structures. In *Proceedings of the November 18-20, 1969, fall joint computer conference*, AFIPS '69 (Fall), pages 27–38, New York, NY, USA, 1969. ACM. 3, 36

[105] B. W. Lampson. Protection. *SIGOPS Oper. Syst. Rev.*, 8(1):18–24, Jan. 1974. 1, 40

[106] B. W. Lampson and D. D. Redell. Experience with processes and monitors in Mesa. *Commun. ACM*, 23(2):105–117, Feb. 1980. 36

[107] M. Leucker and C. Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009. 35, 37

[108] J. Ligatti, L. Bauer, and D. Walker. Run-time enforcement of nonsafety policies. *ACM Trans. Inf. Syst. Secur.*, 12:19:1–19:41, January 2009. 197

[109] R. Longacre. *An Anatomy of Speech Notions.* PdR Press publications in tagmemics. Peter de Ridder Press, 1976. 32

[110] R. Mardare and C. Priami. A logical approach to security in the context of ambient calculus. *Electron. Notes Theor. Comput. Sci.*, 99:3–29, Aug. 2004. 43

[111] E. Marks. *Service-Oriented Architecture (SOA) Governance for the Services Driven Enterprise.* Wiley, 2008. 39

[112] P. Mell and T. Grance. The NIST definition of cloud computing. *National Institute of Standards and Technology*, 53(6):50, 2009. 1

[113] S. Meng, F. Arbab, and C. Baier. Synthesis of Reo circuits from scenario-based interaction specifications. *Science of Computer Programming*, 76(8):651–680, 2011. Special issue on the 7th International Workshop on the Foundations of Coordination Languages and Software Architectures. 33

[114] D. L. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, 39:1482–1493, 1991. 34

[115] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer, 1980. 14, 19, 31, 91

[116] R. Milner. The polyadic π-calculus: a tutorial. The University of Edinburgh, 1991. 14

[117] R. Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, 1999. 13, 14, 17, 19, 69, 72, 78, 81, 82, 83, 91

[118] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, Parts I and II. *Inf. Comput.*, 100(1), 1992. 13, 14, 91

[119] N. H. Minsky. Law governed interaction (LGI): A distributed coordination and control mechanism. *Department of Computer Science, Rutgers University*, 2, 2005. 40

[120] N. H. Minsky. Decentralized governance of networked systems: from access control to interaction control. `http://www.cs.rutgers.edu/~minsky/papers/IC-model.pdf`, 2009. 40

[121] N. H. Minsky. Logic programs, norms and action. chapter Decentralized governance of distributed systems via interaction control, pages 374–400. Springer-Verlag, Berlin, Heidelberg, 2012. 40

[122] N. H. Minsky and J. Leichter. Law-governed Linda as a coordination model. In *Selected papers from the ECOOP'94 Workshop on Models and Languages for Coordination of Parallelism and Distribution, Object-Based Models and Languages for Concurrent Systems*, pages 125–146. Springer-Verlag, 1995. 40

[123] N. H. Minsky and V. Ungureanu. Law-governed interaction: A coordination and control mechanism for heterogeneous distributed systems. *TOSEM, ACM Transactions on Software Engineering and Methodology*, 9:273–305, 2000. 40

[124] U. Montanari and M. Pistore. Concurrent semantics for the π-calculus. *Electronic notes in Computer Science*, (1), 1995. 14

[125] S. Mullender, editor. *Distributed Systems*. Addison-Wesley, 1993. 1

[126] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *In Proc. 17th ACM Symp. on Operating System Principles (SOSP)*, pages 129–142, 1997. 1

[127] D. Nessett. Massively distributed systems: design issues and challenges. pages 8–8, 1999. 1

[128] U. Nestmann and B. C. Pierce. Decoding choice encodings. *Inf. Comput.*, 163(1):1–59, 2000. 22

[129] Ocean Observatories Initiative. `http://www.oceanleadership.org/`. 2, 12, 112, 113, 198, 200

[130] O. Owe, M. Steffen, and A. B. Torjusen. Model testing asynchronously communicating objects using modulo AC rewriting. *ENCS*, 264(3):69–84, 2010. 197

[131] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976. 28, 29

[132] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4(3):455–495, July 1982. 29

[133] C. Palamidessi. Comparing the expressive power of the synchronous and asynchronous pi-calculi. *Mathematical Structures in Computer Science*, 13(5):685–719, 2003. 20, 21, 22

[134] B. C. Pierce. Foundational calculi for programming languages. In A. B. Tucker, editor, *The Computer Science and Engineering Handbook*, pages 2190–2207. CRC Press, 1997. 14

[135] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002. 28

[136] J. Pitt and A. Mamdani. Some remarks on the semantics of FIPA's agent communication language. *Autonomous Agents and Multi-Agent Systems*, 2(4):333–356, Nov. 1999. 32

[137] A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57, November 1977. 29

[138] S. Qadeer and S. Tasiran. Runtime verification of concurrency-specific correctness criteria. *International Journal on Software Tools for Technology Transfer*, 14:291–305, 2012. 34

[139] I.-K. Rhee, J. Lee, J. Kim, E. Serpedin, and Y.-C. Wu. Clock synchronization in wireless sensor networks: An overview. *Sensors*, 9(1):56–85, 2009. 34

[140] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel. Laminar: practical fine-grained decentralized information flow control. *SIGPLAN Not.*, 44(6):63–74, June 2009. 1

[141] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, Feb. 1996. 1

[142] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, Nov. 1997. 34

[143] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3:30–50, 2000. 4, 6, 35, 197

[144] F. B. Schneider, G. Morrisett, and R. Harper. A language-based approach to security. In *INFORMATICS: 10 YEARS BACK, 10 YEARS AHEAD*, pages 86–101. Springer, 2000. 1

[145] Scribble Project homepage. `www.scribble.org`. 38, 198, 199

[146] Scribble development tool site. `http://www.jboss.org/scribble`. 199

[147] P. Selinger. First-order axioms for asynchrony. In *CONCUR*, pages 376–390, 1997. 197

[148] J. Simmonds, Y. Gan, M. Chechik, S. Nejati, B. O'Farrell, E. Litani, and J. Waterhouse. Runtime monitoring of web service conversations. *IEEE T. Services Computing*, 2(3):223–244, 2009. 38

[149] M. Singh. Developing formal specifications to coordinate heterogeneous autonomous agents. In *Multi Agent Systems, 1998. Proceedings. International Conference on*, pages 261–268, jul 1998. 31, 32

[150] I. Smith, P. Cohen, J. Bradshaw, M. Greaves, and H. Holmback. Designing conversation policies using joint intention theory. In *Multi Agent Systems, 1998. Proceedings. International Conference on*, pages 269–276, jul 1998. 32

[151] M. Sridhar and K. W. Hamlen. Model-checking in-lined reference monitors. In *Proceedings of the 11th international conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'10, pages 312–327, Berlin, Heidelberg, 2010. Springer-Verlag. 36

[152] H. Suguri, E. Kodama, and M. Miyazaki. Assuring interoperability in heterogeneous, autonomous and decentralized multi-agent systems. In *Proceedings of the The Sixth International Symposium on Autonomous Decentralized Systems (ISADS'03)*, ISADS '03, pages 17–24, Washington, DC, USA, 2003. IEEE Computer Society. 2

[153] H. Suguri, E. Kodama, M. Miyazaki, and I. Kaji. Assuring interoperability between heterogeneous multi-agent systems with a gateway agent. In *High Assurance Systems Engineering, 2002. Proceedings. 7th IEEE International Symposium on*, pages 167–170, 2002. 2

[154] Y. Tan, L. Yin, Q. Qian, and K. Mori. A biologically inspired assurance definition and specification in heterogeneous autonomous decentralized systems. In *High Assurance Systems Engineering, 2002. Proceedings. 7th IEEE International Symposium on*, pages 63–69, 2002. 2

[155] A. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Pearson Prentice Hall, 2007. 11

[156] UNIFI. International Organization for Standardization ISO 20022 UNIversal Financial Industry message scheme. `http://www.iso20022.org`, 2002. 110

[157] A. Vallecillo, V. T. Vasconcelos, and A. Ravara. Typing the behavior of objects and components using session types. In *FOCLASA'02*, volume 68(3) of *ENTCS*, pages 439–456. Elsevier, 2002. 13

[158] M. Wooldridge. Verifiable semantics for agent communication languages. In *ICMAS*, pages 349–356, 1998. 32

[159] M. Wooldridge. Semantic issues in the verification of agent communication languages. *Autonomous Agents and Multi-Agent Systems*, 3:9–31, 2000. 32

[160] N. Yoshida and M. Hennessy. Assigning types to processes. *Information and Computation*, 172:82–120, 2002. 41

[161] C. G. Zarba. Combining sets with integers. In *FroCos*, pages 103–116, 2002. 193

[162] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. Securing distributed systems with information flow control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 293–308, Berkeley, CA, USA, 2008. USENIX Association. 1

[163] W. Zhang, C. Serban, and N. H. Minsky. Establishing global properties of multi-agent systems via local laws. In *E4MAS'06*, pages 170–183, 2007. 40